Android Developer Fundamentals (Version 2) course

Last updated Tue Sep 11 2018

This course was created by the Google Developer Training team.

- For details about the course, including links to all the concept chapters, apps, and slides, see Android Developer Fundamentals (Version 2).

  developer.android.com/courses/adf-v2

Note: This course uses the terms "codelab" and "practical" interchangeably.

We advise you to use the online version of this course rather than this static PDF to ensure you are using the latest content.

See developer.android.com/courses/adf-v2.

Unit 3: Working in the background

This PDF contains a one-time snapshot of the lessons in **Unit 3: Working in the background.**

Lessons in this unit

Lesson 7.1: AsyncTask

Introduction

A *thread* is an independent path of execution in a running program. When an Android program is launched, the system creates a *main thread*, which is also called the *UI thread*. This UI thread is how your app interacts with components from the Android UI toolkit.

Sometimes an app needs to perform resource-intensive tasks such as downloading files, making database queries, playing media, or computing complex analytics. This type of intensive work can block the UI thread so that the app doesn't respond to user input or draw on the screen. Users may get frustrated and uninstall your app.

To keep the user experience (UX) running smoothly, the Android framework provides a helper class called `AsyncTask`, which processes work off of the UI thread. Using `AsyncTask` to move intensive processing onto a separate thread means that the UI thread can stay responsive.

Because the separate thread is not synchronized with the calling thread, it's called an *asynchronous thread*. An `AsyncTask` also contains a callback that allows you to display the results of the computation back in the UI thread.

In this practical, you learn how to add a background task to your Android app using an `AsyncTask`.

What you should already know

You should be able to:

- Create an `Activity`.
- Add a `TextView` to the layout for the `Activity`.
- Programmatically get the id for the `TextView` and set its content.
- Use `Button` views and their `onClick` functionality.

What you'll learn

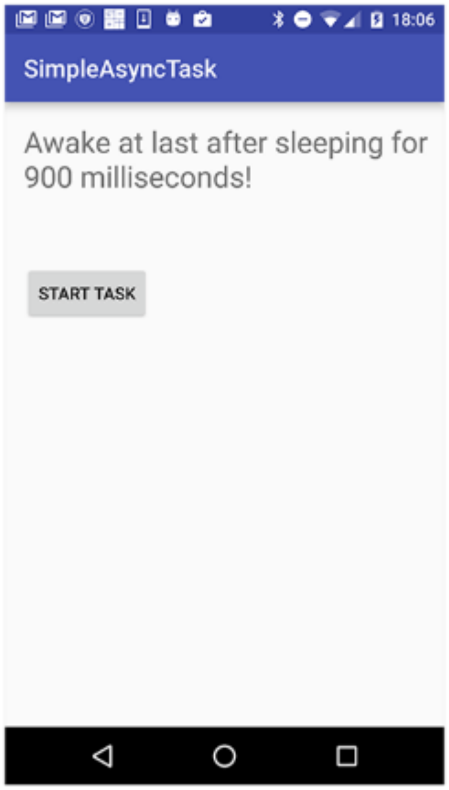
- How to add an `AsyncTask` to your app in order to run a task in the background of your app.
- The drawbacks of using `AsyncTask` for background tasks.

What you'll do

- Create a simple app that executes a background task using an `AsyncTask`.
- Run the app and see what happens when you rotate the device.
- Implement activity instance state to retain the state of a `TextView` message.

App overview

You will build an app that has one `TextView` and one `Button`. When the user clicks the `Button`, the app sleeps for a random amount of time, and then displays a message in the `TextView` when it wakes up. Here's what the finished app looks like:



Awake at last after sleeping for 900 milliseconds!

Task 1: Set up the SimpleAsyncTask project

The SimpleAsyncTask UI contains a `Button` that launches the `AsyncTask`, and a `TextView` that displays the status of the app.

1.1 Create the project and layout

1. Create a new project called SimpleAsyncTask using the **Empty Activity** template. Accept the defaults for all other options.
2. Open the `activity_main.xml` layout file. Click the **Text** tab.

3. Add the `layout_margin` attribute to the top-level `ConstraintLayout`:

```
android:layout_margin="16dp"
```

4. Add or modify the following attributes of the "Hello World!" `TextView` to have these values. Extract the string into a resource.

Attribute	**Value**
`android:id`	`"@+id/textView1`
`android:text`	`"I am ready to start work!"`
`android:textSize`	`"24sp"`

5. Delete the `app:layout_constraintRight_toRightOf` and `app:layout_constraintTop_toTopOf` attributes.

6. Add a `Button` element just under the `TextView`, and give it these attributes. Extract the button text into a string resource.

Attribute	**Value**
`android:id`	`"@+id/button"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:text`	`"Start Task"`
`android:layout_marginTop`	`"24dp"`
`android:onClick`	`"startTask"`
`app:layout_constraintStart_toStartOf`	`"parent"`
`app:layout_constraintTop_toBottomOf`	`"@+id/textView1"`

7. The `onClick` attribute for the button will be highlighted in yellow, because the `startTask()` method is not yet implemented in MainActivity. Place your cursor in the highlighted text, press **Alt + Enter (Option + Enter** on a Mac) and choose **Create 'startTask(View) in 'MainActivity'** to create the method stub in MainActivity.

Solution code for `activity_main.xml`:

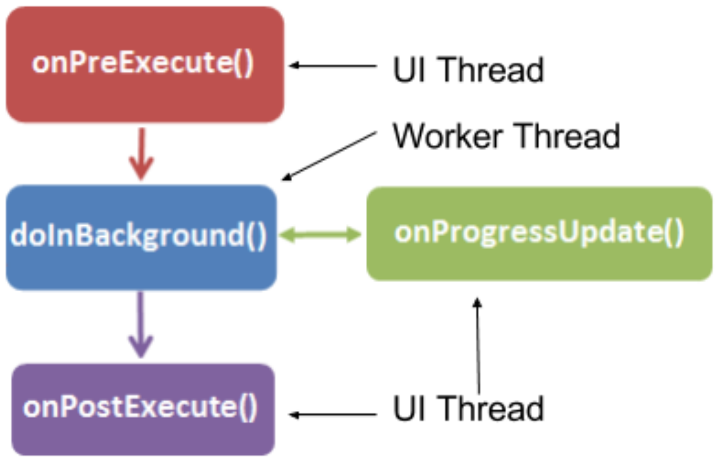
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:layout_margin="16dp"
   tools:context=".MainActivity">

   <TextView
       android:id="@+id/textView1"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="@string/ready_to_start"
       android:textSize="24sp"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toTopOf="parent"/>

   <Button
       android:id="@+id/button"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginTop="24dp"
       android:onClick="startTask"
       android:text="@string/start_task"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toBottomOf="@+id/textView1"/>

</android.support.constraint.ConstraintLayout>
```

Task 2: Create the AsyncTask subclass

`AsyncTask` is an abstract class, which means you must subclass it in order to use it. In this example the `AsyncTask` performs a very simple background task: it sleeps for a random amount of time. In a real app, the background task could perform all sorts of work, from querying a database, to connecting to the internet, to calculating the next Go move to beat the current Go champion.

An `AsyncTask` subclass has the following methods for performing work off of the main thread:

- `onPreExecute()`: This method runs on the UI thread, and is used for setting up your task (like showing a progress bar).
- `doInBackground()`: This is where you implement the code to execute the work that is to be performed on the separate thread.
- `onProgressUpdate()`: This is invoked on the UI thread and used for updating progress in the UI (such as filling up a progress bar)
- `onPostExecute()`: Again on the UI thread, this is used for updating the results to the UI once the AsyncTask has finished loading.



Note: A background or worker thread is any thread which is not the main or UI thread.

When you create an `AsyncTask` subclass, you may need to give it information about the work which it is to perform, whether and how to report its progress, and in what form to return the result. When you create an `AsyncTask` subclass, you can configure it using these parameters:

- Params: The data type of the parameters sent to the task upon executing the `doInBackground()` override method.
- Progress: The data type of the progress units published using the `onProgressUpdated()` override method.
- Result: The data type of the result delivered by the `onPostExecute()` override method.

For example, an `AsyncTask` subclass called `MyAsyncTask` with the following class declaration might take the following parameters:

- A `String` as a parameter in `doInBackground()`, to use in a query, for example.
- An `Integer` for `onProgressUpdate()`, to represent the percentage of job complete
- A `Bitmap` for the result in `onPostExecute()`, indicating the query result.

```
public class MyAsyncTask
   extends AsyncTask <String, Integer, Bitmap>{}
```

In this task you will use an `AsyncTask` subclass to define work that will run in a different thread than the UI thread.

2.1 Subclass the AsyncTask

In this app, the `AsyncTask` subclass you create does not require a query parameter or publish its progress. You will only be using the `doInBackground()` and `onPostExecute()` methods.

1. Create a new Java class called `SimpleAsyncTask` that extends `AsyncTask` and takes three generic type parameters.

   Use `Void` for the params, because this `AsyncTask` does not require any inputs. Use `Void` for the progress type, because the progress is not published. Use a `String` as the result type, because you will update the `TextView` with a string when the `AsyncTask` has completed execution.

```
public class SimpleAsyncTask extends AsyncTask <Void, Void, String>{}
```

Note: The class declaration will be underlined in red, because the required method `doInBackground()` has not yet been implemented.

2. At the top of the class, define a member variable `mTextView` of the type `WeakReference<TextView>`:

```
private WeakReference<TextView> mTextView;
```

3. Implement a constructor for AsyncTask that takes a `TextView` as a parameter and creates a new weak reference for that `TextView`:

```
SimpleAsyncTask(TextView tv) {
       mTextView = new WeakReference<>(tv);
}
```

The `AsyncTask` needs to update the `TextView` in the `Activity` once it has completed sleeping (in the `onPostExecute()` method). The constructor for the class will therefore need a reference to the `TextView` to be updated.

What is the weak reference (the `WeakReference` class) for? If you pass a `TextView` into the `AsyncTask` constructor and then store it in a member variable, that reference to the `TextView` means the `Activity` cannot ever be garbage collected and thus leaks memory, even if the `Activity` is destroyed and recreated as in a device configuration change. This is called creating a *leaky context*, and Android Studio will warn you if you try it.

The weak reference prevents the memory leak by allowing the object held by that reference to be garbage collected if necessary.

2.2 Implement doInBackground()

The `doInBackground()` method is required for your AsyncTask subclass.

1. Place your cursor on the highlighted class declaration, press **Alt + Enter (Option + Enter** on a Mac) and select **Implement methods**. Choose `doInBackground()` and click **OK**. The following method template is added to your class:

```
@Override
protected String doInBackground(Void... voids) {
   return null;
}
```

2. Add code to generate a random integer between 0 and 10. This is the number of milliseconds the task will pause. This is not a lot of time to pause, so multiply that number by 200 to extend that time.

```
Random r = new Random();
int n = r.nextInt(11);

int s = n * 200;
```

3. Add a `try/catch` block and put the thread to sleep.

```
try {
   Thread.sleep(s);
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

4. Replace the existing `return` statement to return the `String` "Awake at last after sleeping for *xx* milliseconds", where *xx* is the number of milliseconds the app slept.

```
return "Awake at last after sleeping for " + s + " milliseconds!";
```

The complete `doInBackground()` method looks like this:

```
@Override
protected String doInBackground(Void... voids) {

   // Generate a random number between 0 and 10
   Random r = new Random();
   int n = r.nextInt(11);

   // Make the task take long enough that we have
   // time to rotate the phone while it is running
   int s = n * 200;

   // Sleep for the random amount of time
   try {
       Thread.sleep(s);
   } catch (InterruptedException e) {
       e.printStackTrace();
   }

   // Return a String result
   return "Awake at last after sleeping for " + s + " milliseconds!";
}
```

2.3 Implement onPostExecute()

When the `doInBackground()` method completes, the return value is automatically passed to the `onPostExecute()` callback.

1. Implement `onPostExecute()` to take a `String` argument and display that string in the TextView:

```
protected void onPostExecute(String result) {
   mTextView.get().setText(result);
}
```

The `String` parameter to this method is what you defined in the third parameter of your `AsyncTask` class definition, and what your `doInBackground()` method returns.

Because `mTextView` is a weak reference, you have to deference it with the `get()` method to get the underlying `TextView` object, and to call `setText()` on it.

Note: You can update the UI in `onPostExecute()` because that method is run on the main thread. You cannot update the `TextView` with the new string in the `doInBackground()` method, because that method is executed on a separate thread.

Task 3: Implement the final steps

3.1 Implement the method that starts the AsyncTask

Your app now has an `AsyncTask` class that performs work in the background (or it would if it didn't call `sleep()` as the simulated work). You can now implement the onClick method for the "Start Task" button to trigger the background task.

1. In the `MainActivity.java` file, add a member variable to store the `TextView`.

```
private TextView mTextView;
```

2. In the `onCreate()` method, initialize `mTextView` to the `TextView` in the layout.

```
mTextView = findViewById(R.id.textView1);
```

3. In the `startTask()` method, Update the `TextView` to show the text "Napping...". Extract that message into a string resource.

```
mTextView.setText(R.string.napping);
```

4. Create an instance of `SimpleAsyncTask`, passing the `TextView` `mTextView` to the constructor. Call `execute()` on that `SimpleAsyncTask` instance.

```
new SimpleAsyncTask(mTextView).execute();
```

Note: The `execute()` method is where you pass comma-separated parameters that are then passed into `doInBackground()` by the system. Because this `AsyncTask` has no parameters, you leave it blank.

Solution code for `MainActivity`:

```
package com.example.android.simpleasynctask;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

/**
* The SimpleAsyncTask app contains a button that launches an AsyncTask
* which sleeps in the asynchronous thread for a random amount of time.
*/
public class MainActivity extends AppCompatActivity {

   // The TextView where we will show results
   private TextView mTextView;

  @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       mTextView = findViewById(R.id.textView1);
   }

   public void startTask(View view) {
       // Put a message in the text view
       mTextView.setText(R.string.napping);

       // Start the AsyncTask.
       new SimpleAsyncTask(mTextView).execute();
   }
}
```

3.2 Implement onSaveInstanceState()

1. Run the app and click the **Start Task** button. How long does the app nap?
2. Click the **Start Task** button again, and while the app is napping, rotate the device. If the background task completes before you can rotate the phone, try again.

Note: You'll notice that when the device is rotated, the `TextView` resets to its initial content, and the `AsyncTask` does not seem able to update the `TextView`.

There are several things going on here:

- When you rotate the device, the system restarts the app, calling `onDestroy()` and then `onCreate()`. The `AsyncTask` will continue running even if the activity is destroyed, but it will lose the ability to report back to the activity's UI. It will never be able to update the TextView that was passed to it, because that particular TextView has also been destroyed.
- Once the activity is destroyed the `AsyncTask` will continue running to completion in the background, consuming system resources. Eventually, the system will run out of resources, and the `AsyncTask` will fail.
- Even without the `AsyncTask`, the rotation of the device resets all of the UI elements to their default state, which for the `TextView` is the default string that you set in the layout file.

For these reasons, an `AsyncTask` is not well suited to tasks which may be interrupted by the destruction of the `Activity`. In use cases where this is critical you can use a different type of background class called an `AsyncTaskLoader` which you will learn about in a later practical.

In order to prevent the `TextView` from resetting to the initial string, you need to save its state. You will now implement `onSaveInstanceState()` to preserve the content of your TextView when the activity is destroyed in response to a configuration change such as device rotation.

Note: Not all uses of `AsyncTask` require you to handle the state of the views on rotation. This app uses a `TextView` to display the results of the app, so preserving the state is useful. In other cases, such as uploading a file, you may not need any persistent information in the UI, so retaining the state is not critical.

3. At the top of the class, add a constant for the key for the current text in the state bundle:

```
private static final String TEXT_STATE = "currentText";
```

4. Override the `onSaveInstanceState()` method in **MainActivity** to preserve the text inside the `TextView` when the activity is destroyed:

```
@Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        // Save the state of the TextView
        outState.putString(TEXT_STATE,
            mTextView.getText().toString());
    }
}
```

5. In `onCreate()`, retrieve the value of the `TextView` from the state bundle when the activity is restored.

```
// Restore TextView if there is a savedInstanceState
if(savedInstanceState!=null){
  mTextView.setText(savedInstanceState.getString(TEXT_STATE));
```

Solution code for MainActivity:

```
package android.example.com.simpleasynctask;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.TextView;

/**
 * The SimpleAsyncTask app contains a button that launches an AsyncTask
 * which sleeps in the asynchronous thread for a random amount of time.
 */
public class MainActivity extends AppCompatActivity {

    //Key for saving the state of the TextView
    private static final String TEXT_STATE = "currentText";

    // The TextView where we will show results
    private TextView mTextView = null;

    /**
```

```
     * Initializes the activity.
     * @param savedInstanceState The current state data
     */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //  Initialize mTextView
        mTextView = (TextView) findViewById(R.id.textView1);

        // Restore TextView if there is a savedInstanceState
        if(savedInstanceState!=null){
           mTextView.setText(savedInstanceState.getString(TEXT_STATE));
        }
    }

    /**`
     * Handles the onCLick for the "Start Task" button. Launches the
AsyncTask
     * which performs work off of the UI thread.
     *
     * @param view The view (Button) that was clicked.
     */
    public void startTask (View view) {
        // Put a message in the text view
        mTextView.setText(R.string.napping);

        // Start the AsyncTask.
        // The AsyncTask has a callback that will update the text view.
        new SimpleAsyncTask(mTextView).execute();
    }


    /**
     * Saves the contents of the TextView to restore on configuration
change.
     * @param outState The bundle in which the state of the activity is
saved       when it is spontaneously destroyed.
     */
    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        // Save the state of the TextView
        outState.putString(TEXT_STATE, mTextView.getText().toString());
    }
}
```

Android Studio project: SimpleAsyncTask

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: The `AsyncTask` class provides another useful override method: `onProgressUpdate()`, which allows you to update the UI while the AsyncTask is running. Use this method to update the UI with the current sleep time. Look to the AsyncTask documentation to see how `onProgressUpdate()` is properly implemented. Remember that in the class definition of your `AsyncTask`, you will need to specify the data type to be used in the `onProgressUpdate()` method.

Summary

- An `AsyncTask` is an abstract Java class that moves intensive processing onto a separate thread.
- `AsyncTask` must be subclassed to be used.
- Avoid doing resource-intensive work in the UI thread, because it can make your UI sluggish or erratic.

- Any code that does not involve drawing the UI or responding to user input should be moved from the UI thread to another, separate thread.
- `AsyncTask` has four key methods: `onPreExecute()`, `doInBackground()`, `onPostExecute()` and `onProgressUpdate()`.
- The `doInBackground()` method is the only method that is actually run on a worker thread. Do not call UI methods in the `doInBackground()` method.
- The other methods of `AsyncTask` run in the UI thread and allow you to call methods of UI components.
- Rotating an Android device destroys and recreates an `Activity`. This can disconnect the UI from the background thread in `AsyncTask`, which will continue to run.

Related concept

The related concept documentation is in 7.1: AsyncTask and AsyncTaskLoader.

Learn more

Android developer documentation:

- Processes and threads overview
- `AsyncTask`

Other resources:

- Android Threading & Background Tasks

Videos:

- Threading Performance 101

- Good AsyncTask Hunting

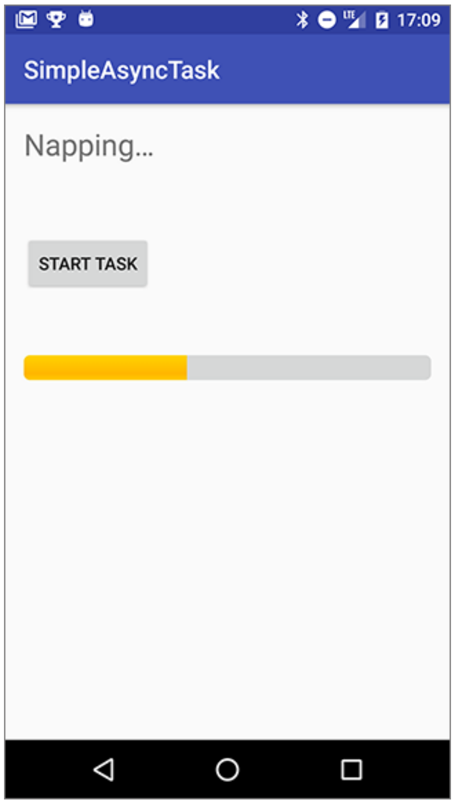
Homework

Build and run an app

Open the SimpleAsyncTask app. Add a `ProgressBar` that displays the percentage of sleep time completed. The progress bar fills up as the AsyncTask thread sleeps from a value of 0 to 100 (percent).

Hint: Break up the sleep time into chunks.

For help, see the `AsyncTask` reference.



Answer these questions

Question 1

For a `ProgressBar`:

1. How do you determine the range of values that a `ProgressBar` can show?
2. How do you change how much of the progress bar is filled in?

Question 2

If an `AsyncTask` is defined as follows:

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long>
```

1. What is the type of the value that is passed to `doInBackground()` in the AsyncTask?
2. What is the type of the value that is passed to the callback that reports the progress of the task?
3. What is the type of the value that is passed to the callback that is executed when the task completes?

Question 3

To report progress of the work executed by an `AsyncTask`, what callback method do you *implement*, and what method do you *call* in your `AsyncTask` subclass?

- Implement `publishProgress()`.
  Call `publishProgress()`.
- Implement `publishProgress()`.
  Call `onProgressUpdate()`.
- Implement `onProgressUpdate()`.
  Call `publishProgress()`.
- Implement `onProgressUpdate()`.
  Call `onProgressUpdate()`.

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- The layout includes a `ProgressBar` that sets the appropriate attributes to determine the range of values.
- The `AsyncTask` breaks the total sleep time into chunks and updates the progress bar after each chunk.
- The `AsyncTask` calls the appropriate method and implements the appropriate callback to update the progress bar.
- The `AsyncTask` needs to know which views to update. Depending on whether the `AsyncTask` is implemented as an inner class or not, the views can either be passed to the constructor of the `AsyncTask` or defined as member variables on the `Activity`.

Lesson 7.2: AsyncTask and AsyncTaskLoader

Introduction

In this practical you use an `AsyncTask` to start a background task that gets data from the internet using a simple REST API. You use the Google APIs Explorer to query the Books API, implement this query in a worker thread using an `AsyncTask`, and display the result in your UI.

Then you reimplement the same background task using `AsyncTaskLoader`, which is a more efficient way to update your UI.

What you should already KNOW

You should be able to:

- Create an activity.

- Add a `TextView` to the layout for the activity.
- Implement `onClick` functionality for a button in your layout.
- Implement an `AsyncTask` and displaying the result in your UI.
- Pass information between activities as extras.

What you'll learn

- How to use the Google APIs Explorer to investigate Google APIs and to view JSON responses to HTTP requests.
- How to use the Google Books API to retrieve data over the internet and keep the UI fast and responsive. You won't learn the Books API in detail—your app will only use the simple book-search function.
- How to parse the JSON results from your API query.
- How to implement an `AsyncTaskLoader` that preserves data on configuration changes.
- How to update your UI using the loader callbacks.

What you'll do

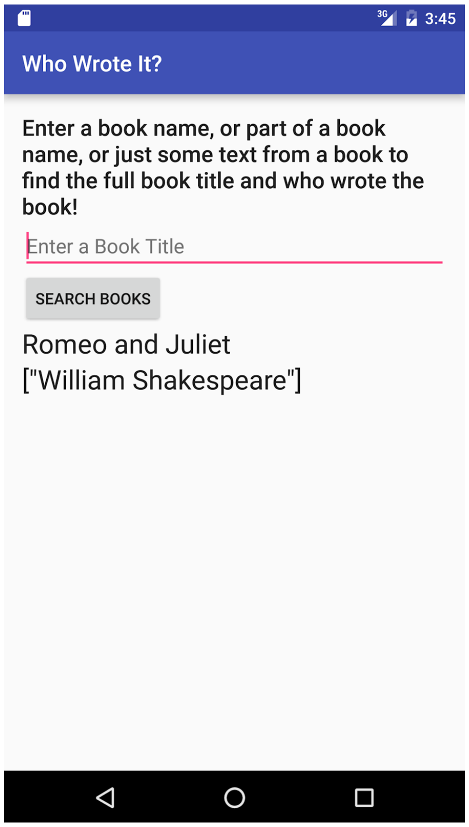
- Use the Google APIs Explorer to learn about the Books API.
- Create the "Who Wrote It?" app, which queries the Books API using a worker thread and displays the result in the UI.
- Modify the "Who Wrote it?" app to use an `AsyncTaskLoader` instead of an `AsyncTask`.

App overview

You will build an app that contains an `EditText` and a `Button`.

- The user enters the name of the book in the `EditText` and taps the button.
- The button executes an `AsyncTask` that queries the Google Books API to find the author and title of the book the user is looking for.
- The results are retrieved and displayed in a `TextView` below the button.

Once the app is working, you modify the app to use `AsyncTaskLoader` instead of the `AsyncTask` class.



Enter a book name, or part of a book name, or just some text from a book to find the full book title and who wrote the book!

Romeo and Juliet
["William Shakespeare"]

Task 1. Explore the Google Books API

In this practical you use the Google Books API to search for information about a book, such as the book's author and title. The Books API provides programmatic access to the Google Book Search service using REST APIs. This is the same service used behind the scenes when you manually execute a search on Google Books. You can use the Google APIs Explorer and Google Book Search in your browser to verify that your Android app is getting the expected results.

1.1 Send a Books API Request

1. Go to the Google APIs Explorer at https://developers.google.com/apis-explorer/.
2. Click **Services** in the left nav, and then **Books API**.
3. Find **books.volumes.list** and click that function name. To find within a page, you can press `Control+F` (`Command+F` on Mac).

   You should see a webpage that lists the parameters of the Books API function that performs book searches.
4. In the `q` field, enter a book name or a partial book name, for example "Romeo". The `q` parameter is the only required field.
5. In the `maxResults` field, enter `10` to limit the results to the top 10 matching books.
6. In the `printType` field, enter `books` to limit the results to books that are in print.
7. Make sure that the **Authorize requests using OAuth 2.0** switch at the top of the form is off.
8. Click **Execute without OAuth** link at the bottom of the form.
9. Scroll down to see the HTTP request and HTTP response.

The HTPP request is a uniform resource identifier (URI). A URI is a string that identifies a resource, and a URL is a certain type of URI that identifies a web resource. For the Books API, the request is a URL. The search parameters that you entered into the form follow the `?` in the URL.

Notice the API `key` field at the end of the URL. For security reasons, when you access a public API, you must obtain an API key and include it in your request. The Books API doesn't require an API key, so you can leave out that portion of the request URI in your app.

1.2 Analyze the Books API response

The response to the query is towards the bottom of the page. The response uses the JSON format, which is a common format for API query responses. In the APIs Explorer web page, the JSON code is nicely formatted so that it is human readable. In your app, the JSON response will be returned from the API service as a single string, and you will need to parse that string to extract the information you need.

The response is made up of name/value pairs that are separated by commas. For example, `"kind": "books#volumes"` is a name/value pair, where `"kind"` is the name and `"books#volumes"` is the value. This is the JSON format.

1. Find the value for the `"title"` name for one book. Notice that this result contains a single value.
2. Find the value for the `"authors"` name for one book. Notice that this result is an array that can contain more than one value.

The book search includes all the books that contain the search string, with multiple objects to represent each book. In this practical, you only return the title and authors of the *first* item in the response.

Task 2. Create the "Who Wrote It?" app

Now that you're familiar with the Books API, it's time to set up the layout of your app.

2.1 Create the project and user interface (UI)

1. Create a new project called "WhoWroteIt", using the Empty Activity template. Accept the defaults for all the other options.
2. Open the `activity_main.xml` layout file. Click the **Text** tab.
3. Add the `layout_margin` attribute to the top-level `ConstraintLayout`:

```
android:layout_margin="16dp"
```

4. Delete the existing `TextView`.
5. Add the following UI elements and attributes to the layout file. Note that the string resources will appear in red; you define those in the next step.

View	**Attributes**	**Values**

`TextView`	`android:layout_width`
	`android:layout_height`
	`android:id`
	`android:text`
	`android:textAppearance`
	
	`app:layout_constraintStart_toStartOf`
	`app:layout_constraintTop_toTopOf`
		`"match_parent"`
		`"wrap_content"`
		`"@+id/instructions"`
		`"@string/instructions"`
		`"@style/TextAppearance.AppCompat.Title"`
		`"parent"`
		`"parent"`

`EditText`	`android:layout_width`
	`android:layout_height`
	`android:id`
	`android:layout_marginTop`
	`android:inputType`
	`android:hint`
	`app:layout_constraintEnd_toEndOf`
	`app:layout_constraintStart_toStartOf`
	`app:layout_constraintTop_toBottomOf`
		`"match_parent"`
		`"wrap_content"`
		`"@+id/bookInput"`
		`"8dp"`
		`"text"`
		`"@string/input_hint"`
		`"parent"`
		`"parent"`
		`"@+id/instructions"`

`Button`	`android:layout_width`
	`android:layout_height`
	`android:id`
	`android:layout_marginTop`
	`android:text`
	`android:onClick`
	`app:layout_constraintStart_toStartOf`
	`app:layout_constraintTop_toBottomOf`
		`"wrap_content"`
		`"wrap_content"`
		`"@+id/searchButton"`
		`"8dp"`
		`"@string/button_text"`
		`"searchBooks"`
		`"parent"`
		`"@+id/bookInput"`

`TextView`	`android:layout_width`
	`android:layout_height`
	`android:id`
	`android:layout_marginTop`
	`android:textAppearance`
	
	`app:layout_constraintStart_toStartOf`
	`app:layout_constraintTop_toBottomOf`
		`"wrap_content"`
		`"wrap_content"`
		`"@+id/titleText"`
		`"16dp"`
		`"@style/TextAppearance.`
		`AppCompat.Headline"`
		`"parent"`
		`"@+id/searchButton"`

`TextView`	`android:layout_width`
	`android:layout_height`
	`android:id`
	`android:layout_marginTop`
	`android:textAppearance`
	
	`app:layout_constraintStart_toStartOf`
	`app:layout_constraintTop_toBottomOf`
		`"wrap_content"`
		`"wrap_content"`
		`"@+id/authorText"`
		`"8dp"`
		`"@style/TextAppearance.`
		`AppCompat.Headline"`
		`"parent"`
		`"@+id/titleText"`

1. In the `strings.xml` file, add these string resources:

```
<string name="instructions">Enter a book name to find out who wrote the
book. </string>
<string name="button_text">Search Books</string>
<string name="input_hint">Book Title</string>
```

1. The `onClick` attribute for the button will be highlighted in yellow, because the `searchBooks()` method is not yet implemented in `MainActivity`. To create the method stub in `MainActivity`, place your cursor in the highlighted text, press `Alt+Enter` (`Option+Enter` on a Mac) and choose **Create 'searchBooks(View) in 'MainActivity'**.

Solution code for `activity_main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:layout_margin="16dp"
   tools:context=".MainActivity">

   <TextView
       android:id="@+id/instructions"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:text="@string/instructions"
       android:textAppearance="@style/TextAppearance.AppCompat.Title"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toTopOf="parent"/>

   <EditText
       android:id="@+id/bookInput"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:layout_marginTop="8dp"
       android:hint="@string/input_hint"
       android:inputType="text"
       app:layout_constraintEnd_toEndOf="parent"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toBottomOf="@+id/instructions"/>

   <Button
```

```
        android:id="@+id/searchButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:onClick="searchBooks"
        android:text="@string/button_text"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/bookInput"/>

    <TextView
        android:id="@+id/titleText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:textAppearance=
           "@style/TextAppearance.AppCompat.Headline"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/searchButton" />

    <TextView
        android:id="@+id/authorText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:textAppearance=
           "@style/TextAppearance.AppCompat.Headline"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/titleText"/>

</android.support.constraint.ConstraintLayout>
```

2.2 Get user input

To query the Books API, you need to get the user input from the `EditText`.

1. In `MainActivity.java`, create member variables for the `EditText`, the author `TextView`, and the title `TextView`.

```
private EditText mBookInput;
private TextView mTitleText;
private TextView mAuthorText;
```

2. Initialize those variables to views in `onCreate()`.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);

   mBookInput = (EditText)findViewById(R.id.bookInput);
   mTitleText = (TextView)findViewById(R.id.titleText);
   mAuthorText = (TextView)findViewById(R.id.authorText);
}
```

3. In the `searchBooks()` method, get the text from the `EditText` view. Convert the text to a `String`, and assign it to a variable.

```
public void searchBooks(View view) {
   // Get the search string from the input field.
   String queryString = mBookInput.getText().toString();
}
```

2.3 Create an empty AsyncTask class

You are now ready to connect to the internet and use the Books API. In this task you create a new `AsyncTask` subclass called `FetchBook` to handle connecting to the network.

Network connectivity can be be sluggish, which can make your app erratic or slow. For this reason, don't make network connections on the UI thread. If you attempt a network connection on the UI thread, the Android runtime might raise a `NetworkOnMainThreadException` to warn you that it's a bad idea.

Instead, use a subclass of `AsyncTask` to make network connections. An `AsyncTask` requires three type parameters: an input-parameter type, a progress-indicator type, and a result type.

1. Create a Java class in your app called `FetchBook`, that extends `AsyncTask`. The generic type parameters for the class will be `<String, Void, String>`. (`String` because the query is a string, `Void` because there is no progress indicator, and `String` because the JSON response is a string.)

```
public class FetchBook extends AsyncTask<String, Void, String> {

}
```

2. Implement the required method, `doInBackground()`. To do this, place your cursor on the red underlined text, press `Alt+Enter` (`Option+Enter` on a Mac) and select **Implement methods**. Choose **doInBackground()** and click **OK**.

Make sure the parameters and return types are correct. (The method takes a variable list of `String` objects and returns a `String`.)

```
@Override
protected String doInBackground(String... strings) {
   return null;
}
```

3. Select **Code** > **Override methods**, or press `Ctrl+O`. Select the **onPostExecute()** method to insert the method definition into the class. The `onPostExecute()` method takes a `String` as a parameter and returns `void`.

```
@Override
protected void onPostExecute(String s) {
   super.onPostExecute(s);

}
```

4. To display the results in the `TextView` objects in `MainActivity`, you must have access to those text views inside the `AsyncTask`. Create `WeakReference` member variables for references to the two text views that show the results.

```
private WeakReference<TextView> mTitleText;
private WeakReference<TextView> mAuthorText;
```

Note: As you learned in the previous practical, you use `WeakReference` objects for these text views (rather than actual `TextView` objects) to avoid leaking context from the `Activity`. The weak references prevent memory leaks by allowing the object held by that reference to be garbage-collected if necessary.

5. Create a constructor for the `FetchBook` class that includes the `TextView` views from `MainActivity`, and initialize the member variables in that constructor.

```
FetchBook(TextView titleText, TextView authorText) {
   this.mTitleText = new WeakReference<>(titleText);
   this.mAuthorText = new WeakReference<>(authorText);
}
```

Solution code for `FetchBook`:

```
public class FetchBook extends AsyncTask<String,Void,String> {
   private WeakReference<TextView> mTitleText;
   private WeakReference<TextView> mAuthorText;

   public FetchBook(TextView mTitleText, TextView mAuthorText) {
      this.mTitleText = new WeakReference<>(titleText);
      this.mAuthorText = new WeakReference<>(authorText);
   }

   @Override
   protected String doInBackground(String... strings) {
       return null;
   }

   @Override
   protected void onPostExecute(String s) {
       super.onPostExecute(s);
   }
}
```

2.4 Create the NetworkUtils class and build the URI

You need to open an internet connection and query the Books API. Because you will probably use this functionality again, you may want to create a utility class with this functionality or develop a useful subclass for your own convenience.

In this task, you write the code for connecting to the internet in a helper class called `NetworkUtils`.

1. Create a new Java class in your app called **`NetworkUtils`**. The `NetworkUtils` class does not extend from any other class.
2. For logging, create a **`LOG_TAG`** variable with the name of the class:

```
private static final String LOG_TAG =
   NetworkUtils.class.getSimpleName();
```

3. Create a static method named **`getBookInfo()`**. The `getBookInfo()` method takes the search term as a **`String`** parameter and returns the JSON `String` response from the API you examined earlier.

```
static String getBookInfo(String queryString){

}
```

4. Create the following local variables in the **`getBookInfo()`** method. You will need these variables for connecting to the internet, reading the incoming data, and holding the response string.

```
HttpURLConnection urlConnection = null;
BufferedReader reader = null;
String bookJSONString = null;
```

5. At the end of the `getBookInfo()` method, return the value of `bookJSONString`.

```
return bookJSONString;
```

6. Add a skeleton `try`/`catch`/`finally` block in `getBookInfo()`, after the local variables and before the `return` statement.

In the `try` block, you'll build the URI and issue the query. In the `catch` block, you'll handle problems with the request. In the `finally` block, you'll close the network connection after you finish receiving the JSON data.

```
try {
   //...
} catch (IOException e) {
      e.printStackTrace();
} finally {
   //...
}
```

7. Create the following member constants at the top of the the `NetworkUtils` class, below the `LOG_TAG` constant:

```
// Base URL for Books API.
private static final String BOOK_BASE_URL =
"https://www.googleapis.com/books/v1/volumes?";
// Parameter for the search string.
private static final String QUERY_PARAM = "q";
// Parameter that limits search results.
private static final String MAX_RESULTS = "maxResults";
// Parameter to filter by print type.
private static final String PRINT_TYPE = "printType";
```

As you saw in the request on the Books API web page, all of the requests begin with the same URI. To specify the type of resource, append query parameters to that base URI. It is common practice to separate all of these query parameters into constants, and combine them using an `Uri.Builder` so they can be reused for different URIs. The `Uri` class has a convenient method, `Uri.buildUpon()`, that returns a `URI.Builder` that you can use.

For this app, you limit the number and type of results returned to increase the query speed. To restrict the query, you will only look for books that are printed.

8. In the `getBookInfo()` method, build your request URI in the `try` block:

```
Uri builtURI = Uri.parse(BOOK_BASE_URL).buildUpon()
       .appendQueryParameter(QUERY_PARAM, queryString)
       .appendQueryParameter(MAX_RESULTS, "10")
       .appendQueryParameter(PRINT_TYPE, "books")
       .build();
```

9. Also inside the `try` block, convert your URI to a URL object:

```
URL requestURL = new URL(builtURI.toString());
```

2.5 Make the request

This API request uses the `HttpURLConnection` class in combination with an `InputStream`, `BufferedReader`, and a `StringBuffer` to obtain the JSON response from the web. If at any point the process fails and `InputStream` or `StringBuffer` are empty, the request returns `null`, signifying that the query failed.

1. In the `try` block of the **`getBookInfo()`** method, open the URL connection and make the request:

```
urlConnection = (HttpURLConnection) requestURL.openConnection();
urlConnection.setRequestMethod("GET");
urlConnection.connect();
```

1. Also inside the `try` block, set up the response from the connection using an `InputStream`, a `BufferedReader` and a `StringBuilder`.

```
// Get the InputStream.
InputStream inputStream = urlConnection.getInputStream();

// Create a buffered reader from that input stream.
reader = new BufferedReader(new InputStreamReader(inputStream));

// Use a StringBuilder to hold the incoming response.
StringBuilder builder = new StringBuilder();
```

1. Read the input line-by-line into the string while there is still input:

```
String line;
while ((line = reader.readLine()) != null) {
   builder.append(line);

   // Since it's JSON, adding a newline isn't necessary (it won't
   // affect parsing) but it does make debugging a *lot* easier
   // if you print out the completed buffer for debugging.
   builder.append("\n");
}
```

Note: The `while` loop adds the incoming line to the builder string in two steps: one step for the line of response data, and one step to add the new line character (`"\n"`).

The new line does not affect JSON parsing of the response, but it makes it a lot easier to debug the response when you view it in the log.

2. At the end of the input, check the string to see if there is existing response content. Return `null` if the response is empty.

```
if (builder.length() == 0) {
   // Stream was empty. No point in parsing.
```

```
    return null;
}
```

3. Convert the `StringBuilder` object to a `String` and store it in the `bookJSONString` variable.

```
bookJSONString = builder.toString();
```

4. In the `finally` block, close both the connection and the `BufferedReader`:

```
finally {
    if (urlConnection != null) {
        urlConnection.disconnect();
    }
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Note: Each time the connection fails for any reason, this code returns `null`. This means that the `onPostExecute()` in the `FetchBook` class has to check its input parameter for a `null` string and let the user know about the failure.

This error handling strategy is simplistic, because the user has no idea *why* the connection failed. A better solution for a production app is to handle each point of failure differently so that the user gets helpful feedback.

5. Just before the final return, print the value of the `bookJSONString` variable to the log.

```
Log.d(LOG_TAG, bookJSONString);
```

6. In `FetchBook`, modify the `doInBackground()` method to call the `NetworkUtils.getBookInfo()` method, passing in the search term that you obtained from the `params` argument passed in by the system. (The search term is the first value in the `strings` array.) Return the result of this method. (This line replaces the `null` return.)

```
return NetworkUtils.getBookInfo(strings[0]);
```

1. In `MainActivity`, add this line to the end of the `searchBooks()` method to launch the background task with the `execute()` method and the query string.

```
    new FetchBook(mTitleText, mAuthorText).execute(queryString);
```

1. Run your app and execute a search. Your app will crash. In Android Studio, click **Logcat** to view the logs and see what is causing the error. You should see the following line:

```
Caused by: java.lang.SecurityException: Permission denied (missing INTERNET
permission?)
```

This error indicates that you have not included the permission to access the internet in your Android manifest. Connecting to the internet introduces security concerns, which is why apps do not have connectivity by default. In the next task you add internet permissions to the manifest.

2.6 Add internet permissions

1. Open the `AndroidManifest.xml` file.
2. Add the following code just before the `<application>` element:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
       android:name="android.permission.ACCESS_NETWORK_STATE" />
```

3. Build and run your app again. In Android Studio, click **Logcat** to view the log. Note that this time, the query runs correctly and the JSON string result is printed to the log.

2.7 Parse the JSON string

Now that you have a JSON response to your query, you must parse the results to extract the information you want to display in your app's UI. Java has classes in its core API help you parse and handle JSON-type data. This process, as well as updating the UI, happen in the `onPostExecute()` method of your `FetchBook` class.

There is a chance that the `doInBackground()` method won't return the expected JSON string. For example, the `try`/`catch` might fail and throw an exception, the network might time out, or other unhandled errors might occur. In those cases, the JSON parsing will fail and will throw an exception. To handle this case, do the JSON parsing in a `try`/`catch` block, and handle the case where incorrect or incomplete data is returned.

1. In the `FetchBook` class, in the `onPostExecute()` method, add a `try`/`catch` block below the call to `super`.

```
try {
   //...
} catch (JSONException e) {
      e.printStackTrace();
}
```

2. Inside the `try` block, use the classes [`JSONObject`] and [`JSONArray`] to obtain the JSON array of items from the result string.

```
JSONObject jsonObject = new JSONObject(s);
JSONArray itemsArray = jsonObject.getJSONArray("items");
```

3. Initialize the variables used for the parsing loop.

```
int i = 0;
String title = null;
String authors = null;
```

4. Iterate through the `itemsArray` array, checking each book for title and author information. With each loop, test to see if both an author and a title are found, and if so, exit the loop. This way, only entries with both a title and author will be displayed.

```
while (i < itemsArray.length() &&
   (authors == null && title == null)) {
    // Get the current item information.
    JSONObject book = itemsArray.getJSONObject(i);
    JSONObject volumeInfo = book.getJSONObject("volumeInfo");

    // Try to get the author and title from the current item,
    // catch if either field is empty and move on.
    try {
        title = volumeInfo.getString("title");
        authors = volumeInfo.getString("authors");
    } catch (Exception e) {
        e.printStackTrace();
    }

    // Move to the next item.
    i++;
}
```

Note: The loop ends at the first match in the response. More responses might be available, but this app only displays the first one.

5. If a matching response is found, update the UI with that response. Because the references to the `TextView` objects are `WeakReference` objects, you have to dereference them using the `get()` method.

```
if (title != null && authors != null) {
    mTitleText.get().setText(title);
    mAuthorText.get().setText(authors);
}
```

6. If the loop has stopped and the result has no items with both a valid author and a valid title, set the title `TextView` to a "no results" string resource and clear the author `TextView`.

```
} else {
    mTitleText.get().setText(R.string.no_results);
    mAuthorText.get().setText("");
}
```

7. In the `catch` block, print the error to the log. Set the title `TextView` to the "no results" string resource, and clear the author `TextView`.

```
} catch (Exception e) {
   // If onPostExecute does not receive a proper JSON string,
   // update the UI to show failed results.
   mTitleText.get().setText(R.string.no_results);
   mAuthorText.get().setText("");
   e.printStackTrace();
}
```

8. Add the `no_results` resource to `strings.xml`:

```
<string name="no_results">"No Results Found"</string>
```

Solution code:

```
@Override
protected void onPostExecute(String s) {
   super.onPostExecute(s);

   try {
       // Convert the response into a JSON object.
       JSONObject jsonObject = new JSONObject(s);
       // Get the JSONArray of book items.
       JSONArray itemsArray = jsonObject.getJSONArray("items");

       // Initialize iterator and results fields.
       int i = 0;
       String title = null;
       String authors = null;

       // Look for results in the items array, exiting
       // when both the title and author
       // are found or when all items have been checked.
       while (i < itemsArray.length() &&
          (authors == null && title == null)) {
           // Get the current item information.
           JSONObject book = itemsArray.getJSONObject(i);
           JSONObject volumeInfo = book.getJSONObject("volumeInfo");

           // Try to get the author and title from the current item,
           // catch if either field is empty and move on.
           try {
               title = volumeInfo.getString("title");
               authors = volumeInfo.getString("authors");
           } catch (Exception e) {
               e.printStackTrace();
           }

           // Move to the next item.
           i++;
       }

       // If both are found, display the result.
       if (title != null && authors != null) {
           mTitleText.get().setText(title);
           mAuthorText.get().setText(authors);
       } else {
           // If none are found, update the UI to
```

```
            // show failed results.
            mTitleText.get().setText(R.string.no_results);
            mAuthorText.get().setText("");
        }

    } catch (Exception e) {
        // If onPostExecute does not receive a proper JSON string,
        // update the UI to show failed results.
        mTitleText.get().setText(R.string.no_results);
        mAuthorText.get().setText("");
    }
}
```

Task 3. Implement UI best practices

You now have a functioning app that uses the Books API to execute a book search. However, a few things do not behave as expected:

- When the user clicks **Search Books**, the keyboard does not disappear. The user has no indication that the query is being executed.
- If there is no network connection, or if the search field is empty, the app still tries to query the API and fails without properly updating the UI.
- If you rotate the screen during a query, the `AsyncTask` becomes disconnected from the `Activity`, and it is not able to update the UI with results.

You fix the first two of these issues in this section, and the last issue in Task 4.

3.1 Hide the keyboard and update the TextView

The user experience of searching is not intuitive. When the user taps the button, the keyboard remains visible, and the user has no way of knowing that the query is in progress.

One solution is to programmatically hide the keyboard and update one of the result text views to read "Loading..." while the query is performed.

1. In `MainActivity`, add the following code to the `searchBooks()` method, after the `queryString` definition. The code hides the keyboard when the user taps the button.

```
InputMethodManager inputManager = (InputMethodManager)
   getSystemService(Context.INPUT_METHOD_SERVICE);
```

```
if (inputManager != null ) {
   inputManager.hideSoftInputFromWindow(view.getWindowToken(),
           InputMethodManager.HIDE_NOT_ALWAYS);
}
```

1. Just beneath the call to execute the `FetchBook` task, add code to change the title `TextView` to a loading message and clear the author `TextView`.

```
new FetchBook(mTitleText, mAuthorText).execute(queryString);
mAuthorText.setText("");
mTitleText.setText(R.string.loading);
```

2. Add the `loading` resource to `strings.xml`:

```
<string name="loading">Loading…</string>
```

3.2 Manage the network state and the empty search field case

Whenever your app uses the network, it needs to handle the possibility that a network connection is unavailable. Before attempting to connect to the network, your app should check the state of the network connection. In addition, it should not try to query the Books API if the user has not entered a query string.

1. In the `searchBooks()` method, use the `ConnectivityManager` and `NetworkInfo` classes to check the network connection. Add the following code after the input manager code that hides the keyboard:

```
ConnectivityManager connMgr = (ConnectivityManager)
           getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = null;
if (connMgr != null) {
   networkInfo = connMgr.getActiveNetworkInfo();
}
```

2. Add a test around the call to the `FetchBook` task and `TextView` updates to ensure that the network connection exists, that the network is connected, and that a query string is available.

```
if (networkInfo != null && networkInfo.isConnected()
           && queryString.length() != 0) {
   new FetchBook(mTitleText, mAuthorText).execute(queryString);
   mAuthorText.setText("");
   mTitleText.setText(R.string.loading);
}
```

3. Add an `else` block to that test. In the `else` block, update the UI with a `no_search_term` error message if there is no term to search for, and a `no_network` error message otherwise.

```
} else {
   if (queryString.length() == 0) {
       mAuthorText.setText("");
       mTitleText.setText(R.string.no_search_term);
   } else {
       mAuthorText.setText("");
       mTitleText.setText(R.string.no_network);
   }
}
```

4. Add the `no_search_term` and `no_network` resources to `strings.xml`:

```
<string name="no_search_term">Please enter a search term</string>
<string name="no_network">Please check your network connection and try
again.</string>
```

Solution code:

```
public void searchBooks(View view) {
   String queryString = mBookInput.getText().toString();

   InputMethodManager inputManager = (InputMethodManager)
           getSystemService(Context.INPUT_METHOD_SERVICE);
   if (inputManager != null ) {
       inputManager.hideSoftInputFromWindow(view.getWindowToken(),
               InputMethodManager.HIDE_NOT_ALWAYS);
   }

   ConnectivityManager connMgr = (ConnectivityManager)
           getSystemService(Context.CONNECTIVITY_SERVICE);
   NetworkInfo networkInfo = null;
   if (connMgr != null) {
       networkInfo = connMgr.getActiveNetworkInfo();
   }

   if (networkInfo != null && networkInfo.isConnected()
           && queryString.length() != 0) {
       new FetchBook(mTitleText, mAuthorText).execute(queryString);
       mAuthorText.setText("");
       mTitleText.setText(R.string.loading);
   } else {
       if (queryString.length() == 0) {
           mAuthorText.setText("");
           mTitleText.setText(R.string.no_search_term);
       } else {
           mAuthorText.setText("");
           mTitleText.setText(R.string.no_network);
       }
   }
}
```

Solution code

The solution code for this practical up to this point is in the Android Studio project WhoWroteIt.

Task 4. Migrate to AsyncTaskLoader

When you use an `AsyncTask` to perform operations in the background, that background thread can't update the UI if a configuration change occurs while the background task is running. To address this situation, use the `AsyncTaskLoader` class.

`AsyncTaskLoader` loads data in the background and reassociates background tasks with the `Activity`, even after a configuration change. With an `AsyncTaskLoader`, if you rotate the device while the task is running, the results are still displayed correctly in the `Activity`.

Why use an `AsyncTask` if an `AsyncTaskLoader` is much more useful? The answer is that it depends on the situation. If the background task is likely to finish before any configuration changes occur, and it's not crucial for the task to update the UI, an `AsyncTask` may be sufficient. The `AsyncTaskLoader` class actually uses an `AsyncTask` behind the scenes to work its magic.

Note: The `AsyncTaskLoader` class is part of the Android platform's `Loader` API, which is a framework to manage loading data into your app in the background. Loaders were deprecated in Android P (API 28) in favor of `ViewModels` and `LiveData`.

The `AsyncTaskLoader` class is still available, but for full backward-compatibility, make sure to use the `AsyncTaskLoader` and other related classes from the Android Support Library.

In this exercise you learn how to use `AsyncTaskLoader` instead of `AsyncTask` to run your Books API query.

4.1 Create an AsyncTaskLoader class

1. To preserve the results of the previous practical, copy the WhoWroteIt project. Rename the copied project "WhoWroteItLoader".
2. Create a class called `BookLoader` that extends `AsyncTaskLoader` with parameterized type `<String>`.

```
import android.support.v4.content.AsyncTaskLoader;

public class BookLoader extends AsyncTaskLoader<String> {

}
```

   Make sure to import the `AsyncTaskLoader` class from the v4 Support Library.

3. Implement the required `loadInBackground()` method. Notice the similarity between this method and the initial `doInBackground()` method from `AsyncTask`.

```
@Nullable
@Override
public String loadInBackground() {
   return null;
}
```

4. Create the constructor for the `BookLoader` class. With your text cursor on the class declaration line, press `Alt+Enter` (`Option+Enter` on a Mac) and select **Create constructor matching super**. This creates a constructor with the `Context` as a parameter.

```
public BookLoader(@NonNull Context context) {
   super(context);
}
```

4.2 Implement required methods

1. Press `Ctrl+O` to open the **Override methods** menu, and select **onStartLoading**. The system calls this method when you start the loader.

```
@Override
protected void onStartLoading() {
   super.onStartLoading();
}
```

2. Inside the `onStartLoading()` method stub, call `forceLoad()` to start the `loadInBackground()` method. The loader will not start loading data until you call the `forceLoad()` method.

```
@Override
protected void onStartLoading() {
   super.onStartLoading();
}
```

3. Create a member variable called `mQueryString` to hold the string for the Books API query. Modify the constructor to take a `String` as an argument and assign it to the `mQueryString` variable.

```
private String mQueryString;

BookLoader(Context context, String queryString) {
   super(context);
   mQueryString = queryString;
}
```

4. In the `loadInBackground()` method, replace the return statement with the following code, which calls the `NetworkUtils`.`getBookInfo()` method with the query string and returns the result:

```
return NetworkUtils.getBookInfo(mQueryString);
```

4.3 Modify MainActivity

The connection between the `AsyncTaskLoader` and the `Activity` that calls it is implemented with the `LoaderManager.LoaderCallbacks` interface. These loader callbacks are a set of methods in the activity that are called by the `LoaderManager` when the loader is being created, when the data has finished loading, and when the loader is reset. The loader callbacks take the results of the task and pass them back to the activity's UI.

In this task you implement the `LoaderManager. LoaderCallbacks` interface in your `MainActivity` to handle the results of the `loadInBackground()` `AsyncTaskLoader` method.

1. In `MainActivity`, add the `LoaderManager.LoaderCallbacks` implementation to the class declaration, parameterized with the `String` type:

```
public class MainActivity extends AppCompatActivity
   implements LoaderManager.LoaderCallbacks<String> {
```

Make sure to import the `LoaderManager.LoaderCallbacks` class from the v4 Support Library.

2. Implement all the required callback methods from the interface. Thi includes `onCreateLoader()`, `onLoadFinished()`, and `onLoaderReset()`. Place your cursor on the class signature line and press `Alt+Enter` (`Option+Enter` on a Mac). Make sure that all the methods are selected and click **OK.**

```
@NonNull
@Override
public Loader<String> onCreateLoader(int id, @Nullable Bundle args) {
   return null;
}

@Override
public void onLoadFinished(@NonNull Loader<String> loader, String data) {

}

@Override
public void onLoaderReset(@NonNull Loader<String> loader) {

}
```

About the required methods:

- `onCreateLoader()` is called when you instantiate your loader.
- `onLoadFinished()` is called when the loader's task finishes. This is where you add the code to update your UI with the results.
- `onLoaderReset()` cleans up any remaining resources.

For this app, you only implement the first two methods. Leave `onLoaderReset()` empty.

3. The `searchBooks()` method is the onClick method for the button. In `searchBooks()`, replace the call to execute the `FetchBook` task with a call to `restartLoader()`. Pass in the query string that you got from the `EditText` in the loader's `Bundle` object:

```
Bundle queryBundle = new Bundle();
queryBundle.putString("queryString", queryString);

getSupportLoaderManager().restartLoader(0, queryBundle, this);
```

The `restartLoader()` method is defined by the `LoaderManager`, which manages all the loaders used in an activity or fragment. Each activity has exactly one `LoaderManager` instance that is responsible for the lifecycle of the `Loaders` that the activity manages.

The `restartLoader()` method takes three arguments:

- A loader `id`, which is useful if you implement more than one loader in your activity.
- An arguments `Bundle` for any data that the loader needs.
- The instance of `LoaderCallbacks` that you implemented in your activity. If you want the loader to deliver the results to the `MainActivity`, specify `this` as the third argument.

4.4 Implement loader callbacks

In this task you implement the `onCreateLoader()` and `onLoadFinished()` callback methods to handle the background task.

1. In `onCreateLoader()`, replace the `return` statement with a statement that returns an instance of the `BookLoader` class. Pass in the context (`this`) and the `queryString` obtained from the passed-in `Bundle`:

```
@NonNull
@Override
public Loader onCreateLoader(int id, @Nullable Bundle args) {
   String queryString = "";

   if (args != null) {
       queryString = args.getString("queryString");
   }

   return new BookLoader(this, queryString);
}
```

1. Copy the code from `onPostExecute()` in your FetchBook class to `onLoadFinished()` in your `MainActivity`. Remove the call to `super.onPostExecute()`. This is the code that parses the JSON result for a match with the query string.
2. Remove all the calls to `get()` for each of the `TextView` objects. Because updating the UI happens in the `Activity` itself, you no longer need weak references to the original views.
3. Replace the argument to the `JSONObject` constructor (the variable `s`) with the parameter `data`.

```
JSONObject jsonObject = new JSONObject(data);
```

4. Run your app. You should have the same functionality as before, but now in a loader! However, when you rotate the device, the view data is lost. That's because when the activity is created (or recreated), the activity doesn't know that a loader is running. To reconnect to the loader, you need an `initLoader()` method in the `onCreate()` of `MainActivity`.
5. Add the following code in `onCreate()` to reconnect to the loader, if the loader already exists:

```
if(getSupportLoaderManager().getLoader(0)!=null){
   getSupportLoaderManager().initLoader(0,null,this);
}
```

If the loader exists, initialize it. You only want to reassociate the loader to the activity if a query has already been executed. In the initial state of the app, no data is loaded, so there is no data to preserve.

1. Run your app again and rotate the device. The loader manager now holds onto your data across device-configuration changes!
2. Remove the `FetchBook` class, because it is no longer used.

Solution code

The solution code for this task is in the Android Studio project WhoWroteItLoader.

Coding challenge

Note: All coding challenges are optional and are not a prerequisite for later lessons.

Challenge 1: Explore the the Books API in greater detail and find a search parameter that restricts the results to books that are downloadable in the EPUB format. Add the parameter to your request and view the results.

Summary

- Tasks that connect to the network should not be executed on the UI thread. The Android runtime usually raises an exception if you attempt network connectivity or file access on the UI thread.
- Use the Books Search API to access Google Books programmatically. An API request to Google Books is in the form of a URL, and the response is a JSON string.
- Use the Google APIs Explorer to explore Google APIs interactively.
- Use `getText()` to retrieve text from an `EditText` view. To convert the text into a simple string, use `toString()`.
- The `Uri.buildUpon()` method returns a `URI.Builder` that you can use to construct URI strings.

- To connect to the internet, you must configure network permission in the Android manifest file:

```
<uses-permission android:name="android.permission.INTERNET" />
```

The [`AsyncTask`](https://developer.android.com/reference/android/os/AsyncTask.html) class lets you run tasks in the background instead of on the UI thread:

- To use an `AsyncTask`, you have to subclass it. The subclass overrides the `doInBackground(Params...)` method. Usually the subclass also overrides the `onPostExecute(Result)` method.
- To start an `AsyncTask`, use `execute()`.
- An `AsyncTask` can't update the UI if the activity that the `AsyncTask` is controlling stops, for example because of a device-configuration change.

When an `AsyncTask` executes, it goes through four steps:

1. `onPreExecute()` runs on the UI thread before the task is executed. This step is normally used to set up the task, for instance by showing a progress bar in the UI.
2. `doInBackground(Params...)` runs on the background thread immediately after `onPreExecute()` finishes. This step performs background computations that can take a long time.
3. `onProgressUpdate(Progress...)` runs on the UI thread after you a call `publishProgress(Progress...)`.
4. `onPostExecute(Result)` runs on the UI thread after the background computation is finished. The result of the computation is passed to `onPostExecute()`.

`AsyncTaskLoader` is the loader equivalent of an `AsyncTask`.

- `AsyncTaskLoader` provides the `loadInBackground()` method, which runs on a separate thread.
- The results of `loadInBackground()` are delivered to the UI thread by way of the `onLoadFinished()` `LoaderManager` callback.
- To create and parse JSON strings, use the built-in Java JSON classes `JSONObject` and `JSONArray`.
- An `AsyncTaskLoader` uses an `AsyncTask` helper class to do work in the background, off the main thread.
- `AsyncTaskLoader` instances are managed by a `LoaderManager`.
- The `LoaderManager` lets you associate a newly created `Activity` with a loader using `getSupportLoaderManager().initLoader()`.

Related concepts

The related concept documentation is in 7.2: Internet connection.

Learn more

Android developer documentation:

- Connect to the network
- Manage network usage
- Loaders
- `AsyncTask`
- `AsyncTaskLoader`

Homework

Build and run an app

Create an app that retrieves and displays the contents of a web page that's located at a URL. The app displays the following:

- A field in which the user enters a URL
- A field such as a menu or spinner that allows the user to choose the protocol (HTTP or HTTPS)

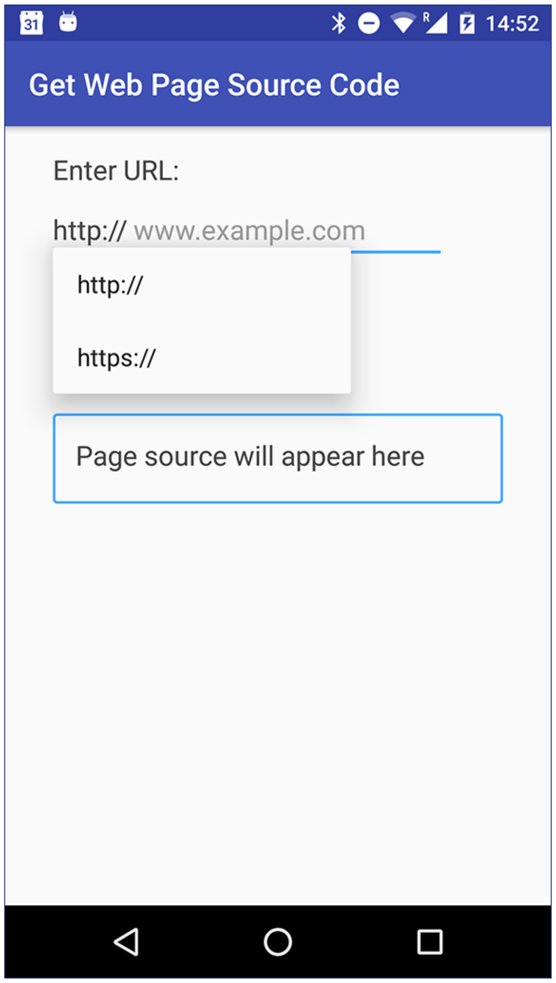
- A button that executes the task when the user taps it
- A scrolling display of the source code of the web page at the URL

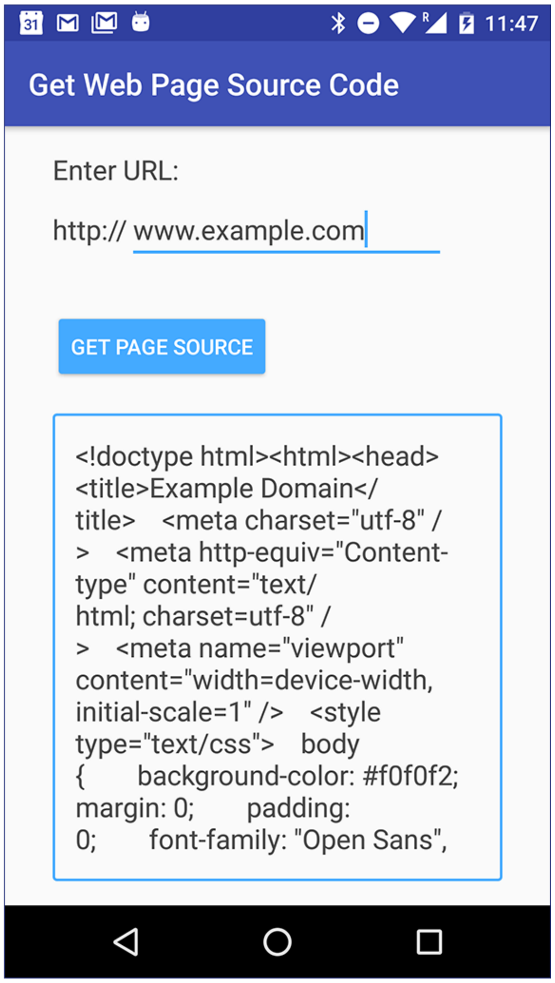
Use an `AsyncTaskLoader` to retrieve the source code of the web page at the URL. You need to implement a subclass of `AsyncTaskLoader`.

If connection to the internet is not available when the user taps the button, the app must show the user an appropriate response. For example, the app might display a message such as "Check your internet connection and try again."

The display must contain a `TextView` in a `ScrollView` that displays the source code, but the exact appearance of the interface is up to you. Your screen can look different from the screenshots below. You can use a pop-up menu, spinner, or checkboxes to allow the user to select HTTP or HTTPS.

The image on the left shows the starting screen, with a pop-up menu for the protocol. The image on the right shows an example of the results of retrieving the page source for given URL.





```
<!doctype html><html><head>
<title>Example Domain</
title>    <meta charset="utf-8" /
>    <meta http-equiv="Content-
type" content="text/
html; charset=utf-8" /
>    <meta name="viewport"
content="width=device-width,
initial-scale=1" />    <style
type="text/css">    body
{       background-color: #f0f0f2;
margin: 0;       padding:
0;       font-family: "Open Sans",
```

Answer these questions

Question 1

What permissions does your app need to connect to the internet?

- `android.permission.CONNECTIVITY`
- `android.permission.INTERNET`
- It doesn't need any special permissions, because all Android apps are allowed to connect to the internet.

Question 2

How does your app check that internet connectivity is available?

In the manifest:

- request `ACCESS_NETWORK_STATE` permission
- request `ALL_NETWORK_STATE` permission
- request `NETWORK_CONNECT` permission

In the code:

- Wrap the code to connect to the internet in a `try`/`catch` block, and catch `NO_NETWORK` errors.
- Use `ConnectivityManager` to check for an active network before connecting to the network.
- Present a dialog to the user reminding them to make sure that internet connectivity is available before they attempt to connect to the internet.

Question 3

Where do you implement the loader callback method that's triggered when the loader finishes executing its task?

- In the `AsyncTaskLoader` subclass. The `AsyncTaskLoader` must implement `LoaderManager.LoaderCallbacks`.
- In the `Activity` that displays the results of the task. The `Activity` must implement `LoaderManager.LoaderCallbacks`.
- In a `Utility` class that extends `Object` and implements `LoaderManager.LoaderCallbacks`.

Question 4

When the user rotates the device, how do `AsyncTask` and `AsyncTaskLoader` behave differently if they are in the process of running a task in the background?

- A running `AsyncTask` becomes disconnected from the activity, but keeps running. A running `AsyncTaskLoader` becomes disconnected from the activity and stops running, preserving system resources.

- A running `AsyncTask` becomes disconnected from the activity and stops running, preserving system resources. A running `AsyncTaskLoader` automatically restarts execution of its task from the beginning. The activity displays the results.
- A running `AsyncTask` becomes disconnected from the activity, but keeps running. A running `AsyncTaskLoader` automatically reconnects to the activity after the device rotation. The activity displays the results.

Question 5

How do you initialize an `AsyncTaskLoader` to perform steps, such as initializing variables, that must be done before the loader starts performing its background task?

- In `onCreateLoader()` in the activity, create an instance of the `AsyncTaskLoader` subclass. In the loader's constructor, perform initialization tasks.
- In `onCreateLoader()` in the activity, create an instance of the `AsyncTaskLoader` subclass. In the loader's `init()` method, perform initialization tasks.
- In the `Activity`, implement `initLoader()` to initialize the loader.
- Perform initialization tasks for the loader at the start of `loadInBackgroud()` in the `Loader`.

Question 6

What methods must an `AsyncTaskLoader` implement?

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- The manifest includes requests for the appropriate permissions.
- Uses a subclass of `AsyncTaskLoader`.
- Responds appropriately if the device can't connect to the internet.
- Combines the protocol and the web page to create a valid URL that the app uses to connect to the internet.
- Implements the required `Loader` callback methods.

- Displays the results of retrieving the source of the web page in a `TextView` in a ScrollView. (It's OK to do it in the same activity, or to start a new activity.)

Lesson 7.3: Broadcast receivers

Introduction

Broadcasts are messages that the Android system and Android apps send when events occur that might affect the functionality of other apps or app components. For example, the Android system sends a *system broadcast* when the device boots up, or when headphones are connected or disconnected. If the wired headset is unplugged, you might like your media app to pause the music.

Your Android app can also broadcast events, for example when new data is downloaded that might interest some other app. Events that your app delivers are called *custom broadcasts*.

In general, you can use broadcasts as a messaging system across apps and outside of the normal user flow.

A broadcast is received by any app or app component that has a *broadcast receiver* registered for that action. `BroadcastReceiver` is the base class for code that receives broadcast intents. To learn more about broadcast receivers, see the Broadcasts overview and the `Intent` reference.

Note: While the `Intent` class is used to send and receive broadcasts, the `Intent` broadcast mechanism is completely separate from intents that are used to start activities.

In this practical, you create an app that responds to a change in the charging state of the device. To do this, your app receives and responds to a system broadcast, and it also sends and receives a custom broadcast.

What you should already KNOW

You should be able to:

- Identify key parts of the `AndroidManifest.xml` file.
- Create Implicit intents.

What you'll learn

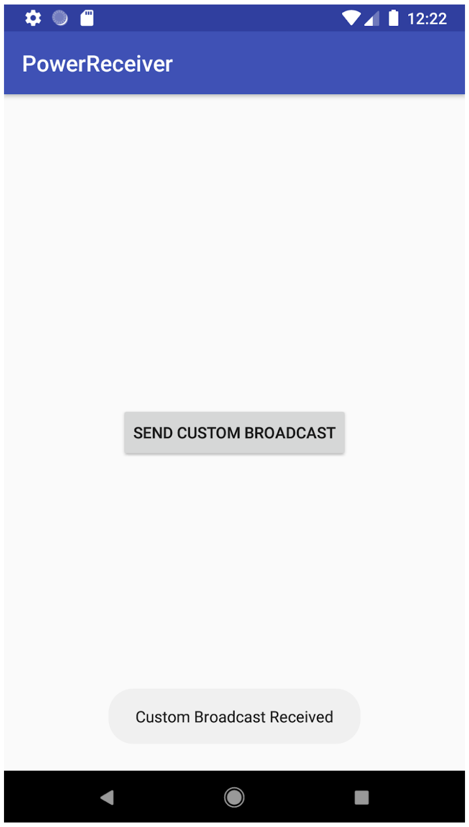
- How to subclass a `BroadcastReceiver` and implement it.
- How to register for system broadcast intents.
- How to create and send custom broadcast intents.

What you'll do

- Subclass a `BroadcastReceiver` to show a toast when a broadcast is received.
- Register your receiver to listen for system broadcasts.
- Send and receive a custom broadcast intent.

App overview

The PowerReceiver app will register a `BroadcastReceiver` that displays a toast message when the device is connected or disconnected from power. The app will also send and receive a custom broadcast to display a different toast message.



Custom Broadcast Received

Task 1. Set up the PowerReceiver project

1.1 Create the project

1. In Android Studio, create a new Java project called **PowerReceiver**. Accept the default options and use the Empty Activity template.
2. To create a new broadcast receiver, select the package name in the Android Project View and navigate to **File > New > Other > Broadcast Receiver**.
3. Name the class **CustomReceiver**. Make sure that **Java** is selected as the source language, and that **Exported** and **Enabled** are selected. **Exported** allows your broadcast receiver to receive broadcasts from outside your app. **Enabled** allows the system to instantiate the receiver.

1.2 Register your receiver for system broadcasts

A *system broadcast* is a message that the Android system sends when a system event occurs. Each system broadcast is wrapped in an `Intent` object:

- The intent's action field contains event details such as `android.intent.action.HEADSET_PLUG`, which is sent when a wired headset is connected or disconnected.
- The intent can contain other data about the event in its extra field, for example a `boolean` extra indicating whether a headset is connected or disconnected.

Apps can register to receive specific broadcasts. When the system sends a broadcast, it routes the broadcast to apps that have registered to receive that particular type of broadcast.

A `BroadcastReceiver` is either a *static receiver* or a *dynamic receiver*, depending on how you register it:

- To register a receiver statically, use the `<receiver>` element in your `AndroidManifest.xml` file. Static receivers are also called *manifest-declared receivers*.

- To register a receiver dynamically, use the app context or activity context. The receiver receives broadcasts as long as the registering context is valid, meaning as long as the corresponding app or activity is running. Dynamic receivers are also called *context-registered receivers*.

For this app, you're interested in two system broadcasts, `ACTION_POWER_CONNECTED` and `ACTION_POWER_DISCONNECTED`. The Android system sends these broadcasts when the device's power is connected or disconnected.

Starting from Android 8.0 (API level 26 and higher), you can't use static receivers to receive most Android system broadcasts, with some exceptions. So for this task, you use dynamic receivers:

1. (Optional) Navigate to your `AndroidManifest.xml` file. Android Studio has generated a `<receiver>` element, but you don't need it, because you can’t use a static receiver to listen for power-connection system broadcasts. Delete the entire `<receiver>` element.
2. In `MainActivity.java`, create a `CustomReceiver` object as a member variable and initialize it.

```
private CustomReceiver mReceiver = new CustomReceiver();
```

Create an intent filter with Intent actions

Intent filters specify the types of intents a component can receive. They are used in filtering out the intents based on `Intent` values like action and category.

1. In `MainActivity.java`, at the end of the `onCreate()` method, create an [`IntentFilter`](https://developer.android.com/reference/android/content/IntentFilter.html) object.

```
IntentFilter filter = new IntentFilter();
```

   When the system receives an `Intent` as a broadcast, it searches the broadcast receivers based on the action value specified in the **`IntentFilter`** object.

2. In `MainActivity.java`, at the end of `onCreate()`, add the actions `ACTION_POWER_CONNECTED` and `ACTION_POWER_DISCONNECTED` to the `IntentFilter` object.

```
filter.addAction(Intent.ACTION_POWER_DISCONNECTED);
filter.addAction(Intent.ACTION_POWER_CONNECTED);
```

Register and unregister the receiver

1. In `MainActivity.java`, at the end of `onCreate()`, register your receiver using the `MainActivity` context. Your receiver is active and able to receive broadcasts as long as your `MainActivity` is running.

```
// Register the receiver using the activity context.
this.registerReceiver(mReceiver, filter);
```

2. In `MainActivity.java`, override the `onDestroy()` method and unregister your receiver. To save system resources and avoid leaks, dynamic receivers must be unregistered when they are no longer needed or before the corresponding activity or app is destroyed, depending on the context used.

```
@Override
   protected void onDestroy() {
       //Unregister the receiver
       this.unregisterReceiver(mReceiver);
       super.onDestroy();
   }
```

1.3 Implement onReceive() in your BroadcastReceiver

When a broadcast receiver intercepts a broadcast that it's registered for, the `Intent` is delivered to the receiver's `onReceive()` method.

In `CustomReceiver.java`, inside the `onReceive()` method, implement the following steps:

1. Delete the entire `onReceive()` method implementation, including the `UnsupportedOperationException` code.
2. Get the `Intent` action from the `intent` parameter and store it in a `String` variable called `intentAction`.

```
@Override
public void onReceive(Context context, Intent intent) {
   String intentAction = intent.getAction();
}
```

3. Create a `switch` statement with the `intentAction` string. (Before using `intentAction`, do a `null` check on it.) Display a different toast message for each action your receiver is registered for.

```
if (intentAction != null) {
   String toastMessage = "unknown intent action";
   switch (intentAction){
       case Intent.ACTION_POWER_CONNECTED:
           toastMessage = "Power connected!";
           break;
       case Intent.ACTION_POWER_DISCONNECTED:
           toastMessage = "Power disconnected!";
           break;
   }
```

```
    //Display the toast.
}
```

4. After the `switch` statement, add code to display a toast message for a short time:

```
Toast.makeText(context, toastMessage, Toast.LENGTH_SHORT).show();
```

5. Run your app. After the app is running, connect or disconnect your device's power supply. A `Toast` is displayed each time you connect or disconnect the power supply, as long as your `Activity` is running.

Note: If you're using an emulator, toggle the power connection state by selecting the ellipses icon for the menu. Select **Battery** in the left bar, then use the **Charger connection** setting.

Task 2. Send and receive a custom broadcast

In addition to responding to system broadcasts, your app can send and receive custom broadcasts. Use a custom broadcast when you want your app to take an action without launching an activity, for example when you want to let other apps know that data has been downloaded to the device.

Android provides three ways for your app to send custom broadcasts:

- *Normal broadcasts* are asynchronous. Receivers of normal broadcasts run in an undefined order, often at the same time. To send a normal broadcast, create a broadcast intent and pass it to `sendBroadcast(Intent)`.
- *Local broadcasts* are sent to receivers that are in the same app as the sender. To send a local broadcast, create a broadcast intent and pass it to `LocalBroadcastManager.sendBroadcast`.
- *Ordered broadcasts* are delivered to one receiver at a time. As each receiver executes, it can propagate a result to the next receiver, or it can cancel the broadcast so that the broadcast is

not passed to other receivers. To send an ordered broadcast, create a broadcast intent and pass it to `sendOrderedBroadcast(Intent, String)`.

This practical doesn't cover ordered broadcasts, but for more information about them, see Sending broadcasts.

The broadcast message is wrapped in an `Intent` object. The `Intent` action string must provide the app's Java package name syntax and uniquely identify the broadcast event.

For a custom broadcast, you define your own `Intent` action (a unique string). You can create `Intent` objects with custom actions and broadcast them yourself from your app using one of the methods above. The broadcasts are received by apps that have a `BroadcastReceiver` registered for that action.

In this task, you add a button to your activity that sends a local broadcast intent. Your receiver registers the broadcast intent and displays the result in a toast message.

2.1 Define your custom broadcast action string

Both the sender and receiver of a custom broadcast must agree on a unique action string for the `Intent` being broadcast. It's a common practice to create a unique action string by prepending your action name with your app's package name.

One of the simplest ways to get your app's package name is to use `BuildConfig.APPLICATION_ID`, which returns the `applicationId` property's value from your module-level `build.gradle` file.

1. Create a constant member variable in both your `MainActivity` and your `CustomReceiver` class. You'll use this variable as the broadcast `Intent` action.

```
private static final String ACTION_CUSTOM_BROADCAST =
BuildConfig.APPLICATION_ID + ".ACTION_CUSTOM_BROADCAST";
```

Important: Although intents are used both for sending broadcasts and starting activities with `startActivity(Intent)`, these actions are completely unrelated. Broadcast receivers can't see or capture an `Intent` that's used to start an activity. Likewise, when you broadcast an `Intent`, you can't use that `Intent` to find or start an activity.

2.2 Add a "Send Custom Broadcast" button

1. In your `activity_main.xml` layout file, replace the Hello World `Textview` with a `Button` that has the following attributes:

```
<Button
   android:id = "@+id/sendBroadcast"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Send Custom Broadcast"
   android:onClick="sendCustomBroadcast"
   app:layout_constraintBottom_toBottomOf="parent"
   app:layout_constraintLeft_toLeftOf="parent"
   app:layout_constraintRight_toRightOf="parent"
   app:layout_constraintTop_toTopOf="parent" />
```

2. Extract the string resource.

The **`sendCustomBroadcast()`** method will be the click-event handler for the button. To create a stub for `sendCustomBroadcast()` in Android Studio:

1. Click the yellow highlighted `sendCustomBroadcast` method name. A red light bulb appears on the left.
2. Click the red light bulb and select **Create 'sendCustomBroadcast(View)' in 'MainActivity'**.

2.3 Implement sendCustomBroadcast()

Because this broadcast is meant to be used solely by your app, use `LocalBroadcastManager` to manage the broadcast. `LocalBroadcastManager` is a class that allows you to register for and send broadcasts that are of interest to components within your app.

By keeping broadcasts local, you ensure that your app data isn't shared with other Android apps. Local broadcasts keep your information more secure and maintain system efficiency.

In `MainActivity.java`, inside the **`sendCustomBroadcast()`** method, implement the following steps:

1. Create a new `Intent`, with your custom action string as the argument.

```
Intent customBroadcastIntent = new Intent(ACTION_CUSTOM_BROADCAST);
```

2. After the custom `Intent` declaration, send the broadcast using the `LocalBroadcastManager` class:

```
LocalBroadcastManager.getInstance(this).sendBroadcast(customBroadcastIntent)
;
```

2.4 Register and unregister your custom broadcast

Registering for a local broadcast is similar to registering for a system broadcast, which you do using a dynamic receiver. For broadcasts sent using `LocalBroadcastManager`, static registration in the manifest is not allowed.

If you register a broadcast receiver dynamically, you must unregister the receiver when it is no longer needed. In your app, the receiver only needs to respond to the custom broadcast when the app is running, so you can register the action in `onCreate()` and unregister it in `onDestroy()`.

1. In `MainActivity.java`, inside `onCreate()` method, get an instance of `LocalBroadcastManager` and register your receiver with the custom `Intent` action:

```
LocalBroadcastManager.getInstance(this)
             .registerReceiver(mReceiver,
                          new IntentFilter(ACTION_CUSTOM_BROADCAST));
```

2. In `MainActivity.java`, inside the `onDestroy()` method, unregister your receiver from the `LocalBroadcastManager`:

```
LocalBroadcastManager.getInstance(this)
          .unregisterReceiver(mReceiver);
```

2.5 Respond to the custom broadcast

1. In `CustomReceiver.java`, inside the `onReceive()` method, add another `case` statement in the `switch` block for the custom `Intent` action. Use `"Custom Broadcast Received"` as the text for the toast message.

```
case ACTION_CUSTOM_BROADCAST:
   toastMessage = "Custom Broadcast Received";
   break;
```

2. Extract the string resource.
3. Run your app and tap the **Send Custom Broadcast** button to send a custom broadcast. Your receiver (`CustomReceiver`) displays a toast message.

That's it! Your app delivers a custom broadcast and is able to receive both system and custom broadcasts.

Solution code

Android Studio project: PowerReceiver

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: If you were developing a music-player app, your app might need to play or pause music when the user connected or disconnected a wired headset. To implement this functionality, you need a broadcast receiver that responds to wired headset events. Implement a broadcast receiver that shows a toast message when a wired headset is connected or disconnected.

Hint: You need to register for the `ACTION_HEADSET_PLUG` action. Because this is a system broadcast action, you can't register for it statically. Instead, register your receiver dynamically by using the context with `Context.registerReceiver()`:

```
IntentFilter filter = new IntentFilter(Intent.ACTION_HEADSET_PLUG);
this.registerReceiver(mReceiver, filter);
```

You must also unregister the receiver when you no longer need it:

```
unregisterReceiver(mReceiver);
```

Summary

- Broadcast receivers are fundamental components of an Android app.
- Broadcast receivers can receive broadcasts sent by the system or by apps.
- The `Intent` used in the broadcast mechanism is completely different from intents used to start activities.
- To process the incoming `Intent` associated with a broadcast, you subclass the `BroadcastReceiver` class and implement `onReceive()`.
- You can register a broadcast receiver in the Android manifest file or programmatically.
- Local broadcasts are private to your app. To register and send local broadcasts, use `LocalBroadcastManager`. Local broadcasts don't involve interprocess communication, which makes them efficient. Using local broadcasts can also protect your app against some security issues, because data stays inside your app.
- To create unique `Intent` action names for broadcasts, a common practice is to prepend the action name with your package name.
- If your app targets API level 26 or higher, you cannot use the manifest to declare a receiver for most implicit broadcasts. (*Implicit broadcasts*, which include most system broadcasts, are broadcasts that don't target your app.) A few implicit broadcasts are exceptions. However, you can use dynamic receivers to receive all broadcasts.

Related concept

The related concept documentation is in 7.3: Broadcasts.

Learn more

Android developer documentation:

- Intents and Intent Filters
- Broadcasts overview
- `BroadcastReceiver`
- Implicit Broadcast Exceptions

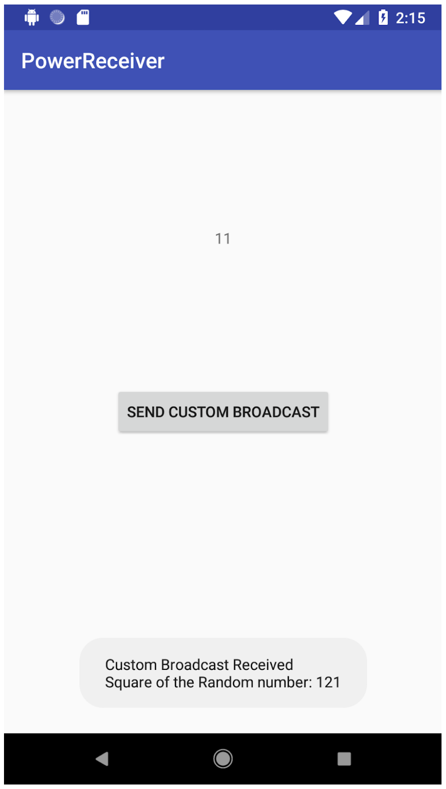
Homework

Update an app

Extend the PowerReceiver app that you created in the practical.

1. Send extra data to your local custom broadcast receiver. To do this, generate a random integer between 1 and 20. Add the number to the `extra` field of the `Intent` before sending the local custom broadcast.

2. In your receiver, extract the integer data from the `Intent`. In the toast message, display the square of the random number.



Custom Broadcast Received
Square of the Random number: 121

Answer these questions

Question 1

What is a system broadcast?

- A message that your app sends and receives when an event of interest occurs in the app.
- A message that is sent from an app to a different component of the same app.
- A message that the Android system sends when a system event occurs.
- A message that the Android system receives when an event of interest occurs in your app.

Question 2

Which pair of methods do you use to register and unregister your broadcast receiver dynamically?

- `registerBroadcast()` and `unRegisterBroadcast()`.

- `registerComponentCallbacks()` and `unRegisterComponentCallbacks()`.
- `registerBroadcastReceiver()` and `unRegisterBroadcastReceiver()`.
- `registerReceiver()` and `unRegisterReceiver()`.

Question 3

Which of the following are true?

- Broadcast receivers can't see or capture the intents used to start an activity.
- Using a broadcast intent, you can't find or start an activity.
- You can use a broadcast intent to start an activity.
- You can receive the intent used to start activity in your broadcast receiver.

Question 4

Which class is used to mitigate the security risks of broadcast receivers when the broadcasts are not cross-application (that is, when broadcasts are sent and received by the same app)?

- `SecureBroadcast`
- `LocalBroadcastManager`
- `OrderedBroadcast`
- `SecureBroadcastManager`

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- The app generates a random integer and sends the integer as an `Intent` extra in the `LocalBroadcast`.
- In the receiver's `onReceive()` method, the random integer data is extracted from the `Intent`, and the integer's square is displayed in a toast message.

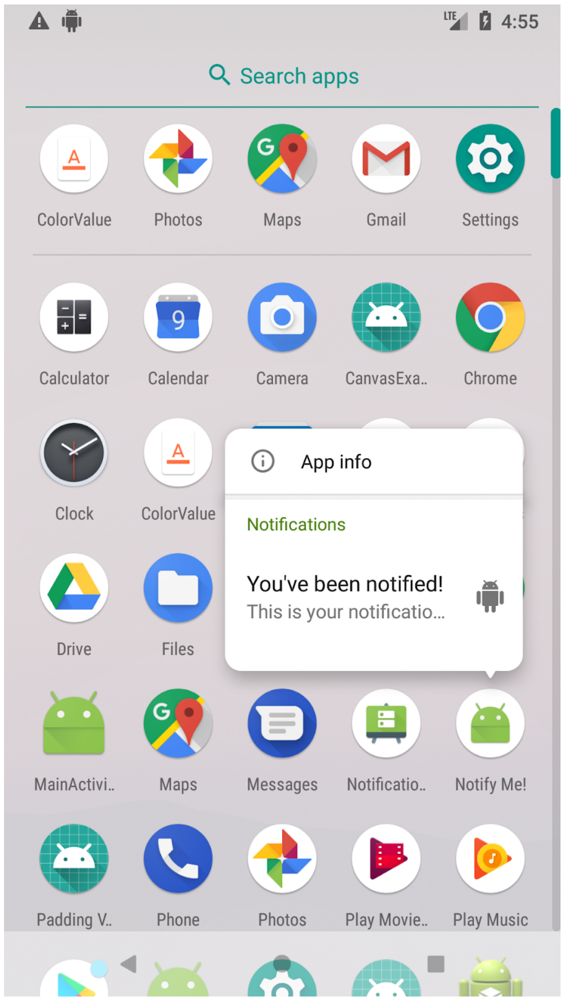
Lesson 8.1: Notifications

Introduction

Sometimes you want your app to show information to users even when the app isn't running in the foreground. For example, you might want to let users know that new content is available, or that their favorite sports team just scored a goal in a game. The Android notification framework provides a way for your app to notify users even when the app is not in the foreground.

A *notification* is a message that your app displays to the user outside of your app's normal UI. Notifications appear as icons in the device's notification area, which is in the status bar. To see the details of a notification, the user opens the *notification drawer*, for example by swiping down on the status bar. The notification area and the notification drawer are system-controlled areas that the user can view at any time.

On devices running Android 8.0 and higher, when your app has a new notification to show to the user, your app icon automatically shows a *badge.* (Badges are also called *notification dots*). When the user long-presses the app icon, the notification appears above the app icon, as shown in the screenshot below.



You've been notified!
This is your notificatio...

In this practical you create an app that triggers a notification when the user taps a button in your app. The user can update the notification or cancel it.

What you should already know

You should be able to:

- Implement the `onClick()` method for buttons.

- Create implicit intents.
- Send custom broadcasts.
- Use broadcast receivers.

What you'll learn

- How to create a notification using the notification builder.
- How to use pending intents to respond to notification actions.
- How to update or cancel existing notifications.

What you'll do

- Create an app that sends a notification when the user taps a button in the app.
- Update the notification from a button in your app, and from an action button that's inside the notification.

App overview

Notify Me! is an app that lets the user trigger, update, and cancel a notification using the three buttons shown in the screenshots below. While you create the app, you'll experiment with notification styles, actions, and priorities.

Task 1: Create a basic notification

1.1 Create the project

1. In Android Studio, create a new project called "Notify Me!" Accept the default options, and use the Empty Activity template.
2. In your `activity_main.xml` layout file, replace the default `TextView` with a button that has the following attributes:

```
<Button
    android:id="@+id/notify"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Notify Me!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Do the following steps in the `MainActivity.java` file:

1. Create a member variable for the **Notify Me!** button:

```
private Button button_notify;
```

1. Create a method stub for the `sendNotification()` method:

```
public void sendNotification() {}
```

2. In the `onCreate()` method, initialize the **Notify Me!** button and create an `onClickListener` for it. Call `sendNotification()` from the `onClick` method:

```
button_notify = findViewById(R.id.notify);
button_notify.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View view) {
       sendNotification();
```

```
    }
});
```

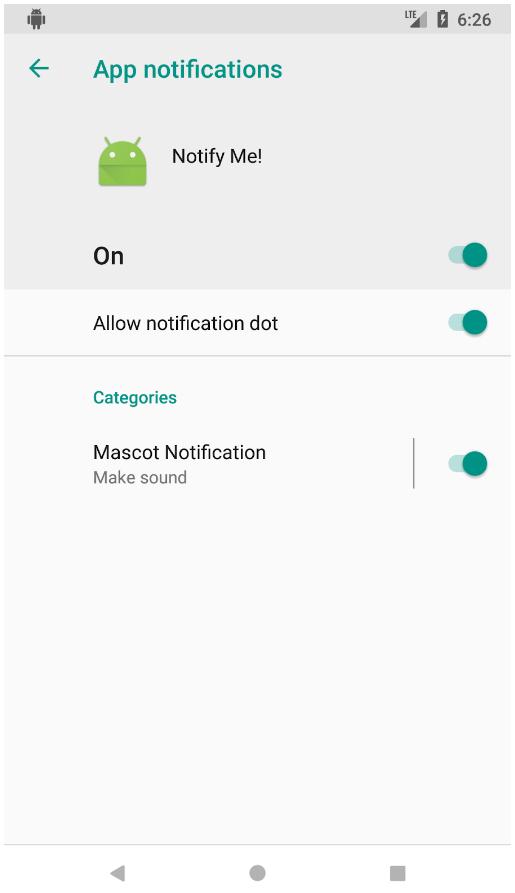
1.2 Create a notification channel

In the Settings app on an Android-powered device, users can adjust the notifications they receive. Starting with Android 8.0 (API level 26), your code can assign each of your app's notifications to a user-customizable notification channel:

- Each *notification channel* represents a type of notification.
- In your code, you can group several notifications in each notification channel.
- For each notification channel, your app sets *behavior* for the channel, and the behavior is applied to all the notifications in the channel. For example, your app might set the notifications in a channel to play a sound, blink a light, or vibrate.
- Whatever behavior your app sets for a notification channel, the user can change that behavior, and the user can turn off your app's notifications altogether.

On Android-powered devices running Android 8.0 (API level 26) or higher, notification channels that you create in your app appear as **Categories** under **App notifications** in the device Settings app.

For example, in the screenshot below of a device running Android 8.0, the Notify Me! app has one notification channel, **Mascot Notification**.


6:26

App notifications

Notify Me!

On

Allow notification dot

Categories

Mascot Notification
Make sound

When your app targets Android 8.0 (API level 26), to display notifications to your users you *must* implement at least one notification channel. To display notifications on lower-end devices, you're not required to implement notification channels. However, it's good practice to always do the following:

- Target the latest available SDK.
- Check the device's SDK version in your code. If the SDK version is 26 or higher, build notification channels.

If your `targetSdkVersion` is set to 25 or lower, when your app runs on Android 8.0 (API level 26) or higher, it behaves the same as it would on devices running Android 7.1 (API level 25) or lower.

Create a notification channel:

1. In `MainActivity`, create a constant for the notification channel ID. Every notification channel must be associated with an ID that is unique within your package. You use this channel ID later, to post your notifications.

```
private static final String PRIMARY_CHANNEL_ID =
"primary_notification_channel";
```

2. The Android system uses the [`NotificationManager`]() class to deliver notifications to the user. In `MainActivity.java`, create a member variable to store the `NotificationManager` object.

```
private NotificationManager mNotifyManager;
```

3. In `MainActivity.java`, create a `createNotificationChannel()` method and instantiate the `NotificationManager` inside the method.

```
public void createNotificationChannel()
{
     mNotifyManager = (NotificationManager)
            getSystemService(NOTIFICATION_SERVICE);
}
```

4. Create a notification channel in the `createNotificationChannel()` method. Because notification channels are only available in API 26 and higher, add a condition to check for the device's API version.

```
public void createNotificationChannel() {
mNotifyManager = (NotificationManager)
       getSystemService(NOTIFICATION_SERVICE);
     if (android.os.Build.VERSION.SDK_INT >=
                                 android.os.Build.VERSION_CODES.O) {
     // Create a NotificationChannel
     }
}
```

5. Inside the `if` statement, construct a [`NotificationChannel`]() object and use `PRIMARY_CHANNEL_ID` as the channel `id`.

6. Set the channel `name`. The `name` is displayed under notification **Categories** in the device's user-visible Settings app.

7. Set the importance to `IMPORTANCE_HIGH`. (For the complete list of notification importance constants, see the `NotificationManager` documentation.)

```
// Create a NotificationChannel
NotificationChannel notificationChannel = new
NotificationChannel(PRIMARY_CHANNEL_ID,
       "Mascot Notification", NotificationManager
       .IMPORTANCE_HIGH);
```

8. In `createNotificationChannel()`, inside the `if` statement, configure the `notificationChannel` object's initial settings. For example, you can set the notification light color, enable vibration, and set a description that's displayed in the device's Settings app. You can also configure a notification alert sound.

```
notificationChannel.enableLights(true);
notificationChannel.setLightColor(Color.RED);
notificationChannel.enableVibration(true);
notificationChannel.setDescription("Notification from Mascot");
mNotifyManager.createNotificationChannel(notificationChannel);
```

1.3 Build your first notification

Notifications are created using the `NotificationCompat.Builder` class, which allows you to set the content and behavior of the notification. A notification can contain the following elements:

- Icon (required), which you set in your code using the `setSmallIcon()` method.
- Title (optional), which you set using `setContentTitle()`.
- Detail text (optional), which you set using `setContentText()`.

To create the required notification icon:

1. In Android Studio, go to **File > New > Image Asset**.
2. From the **Icon Type** drop-down list, select **Notification Icons**.
3. Click the icon next to the **Clip Art** item to select a Material Design icon for your notification. For this app, use the Android icon.

4. Rename the resource `ic_android` and click **Next** and **Finish**. This creates drawable files with different resolutions for different API levels.

To build your notification and display it:

1. You need to associate the notification with a notification ID so that your code can update or cancel the notification in the future. In `MainActivity.java`, create a constant for the notification ID:

```
private static final int NOTIFICATION_ID = 0;
```

2. In `MainActivity.java`, at the end of the `onCreate()` method, call `createNotificationChannel()`. If you miss this step, your app crashes!
3. In `MainActivity.java`, create a helper method called `getNotificationBuilder()`. You use `getNotificationBuilder()` later, in the `NotificationCompat.Builder` object. Android Studio will show an error about the missing return statement, but you'll fix that soon.

```
private NotificationCompat.Builder getNotificationBuilder(){}
```

4. Inside the `getNotificationBuilder()` method, create and instantiate the notification builder. For the notification channel ID, use `PRIMARY_CHANNEL_ID`. If a popup error is displayed, make sure that the `NotificationCompat` class is imported from the v4 Support Library.

```
NotificationCompat.Builder notifyBuilder = new
NotificationCompat.Builder(this, PRIMARY_CHANNEL_ID);
```

5. Inside the `getNotificationBuilder()` method, add the title, text, and icon to the builder, as shown below. At the end, return the `Builder` object.

```
NotificationCompat.Builder notifyBuilder = new
NotificationCompat.Builder(this, PRIMARY_CHANNELID)
        .setContentTitle("You've been notified!")
        .setContentText("This is your notification text.")
        .setSmallIcon(R.drawable.ic_android);
return notifyBuilder;
```

Now you can finish the `sendNotification()` method that sends the notification:

1. In `MainActivity.java`, inside the `sendNotification()` method, use `getNotificationBuilder()` to get the `Builder` object.
2. Call `notify()` on the `NotificationManager`:

```
NotificationCompat.Builder notifyBuilder = getNotificationBuilder();
mNotifyManager.notify(NOTIFICATION_ID, notifyBuilder.build());
```

1. Run your app. The **Notify Me!** button issues a notification, and the icon appears in the status bar. However, the notification is missing an essential feature: nothing happens when you tap it. You add that functionality in the next task.

1.4 Add a content intent and dismiss the notification

Content intents for notifications are similar to the intents you've used throughout this course. Content intents can be explicit intents to launch an activity, implicit intents to perform an action, or broadcast intents to notify the system of a system event or custom event.

The major difference with an `Intent` that's used for a notification is that the `Intent` must be wrapped in a `PendingIntent`. The `PendingIntent` allows the Android notification system to perform the assigned action on behalf of your code.

In this step you update your app so that when the user taps the notification, your app sends a content intent that launches the `MainActivity`. (If the app is open and active, tapping the notification will not have any effect.)

1. In `MainActivity.java`, in the beginning of the `getNotificationBuilder()`, create an explicit intent method to launch the `MainActivity`:

```
Intent notificationIntent = new Intent(this, MainActivity.class);
```

2. Inside `getNotificationBuilder()`, after the `notificationIntent` declaration, use the `getActivity()` method to get a `PendingIntent`. Pass in the notification ID constant for the `requestCode` and use the `FLAG_UPDATE_CURRENT` flag.

   By using a `PendingIntent` to communicate with another app, you are telling that app to execute some predefined code at some point in the future. It's like the other app can perform an action on behalf of your app.

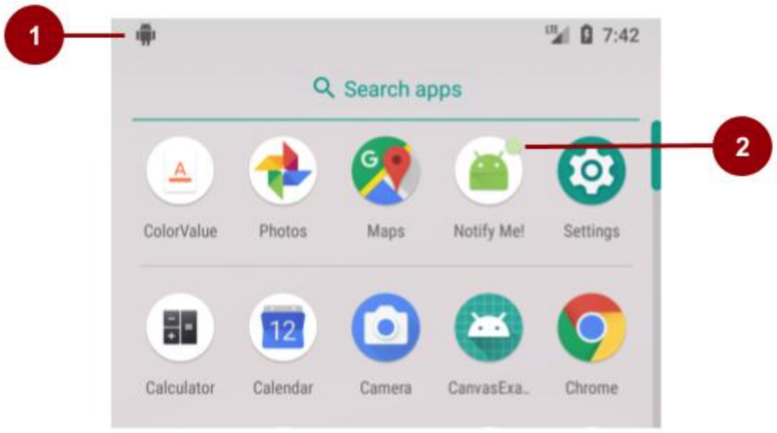
```
    PendingIntent notificationPendingIntent =
PendingIntent.getActivity(this,
        NOTIFICATION_ID, notificationIntent,
PendingIntent.FLAG_UPDATE_CURRENT);
```

3. Use the `setContentIntent()` method from the `NotificationCompat.Builder` class to set the content intent. Inside `getNotificationBuilder()`, call `setContentIntent()` in the code that's building the notification. Also set auto-cancel to `true`:

```
.setContentIntent(notificationPendingIntent)
.setAutoCancel(true)
```

Setting auto-cancel to `true` closes the notification when user taps on it.

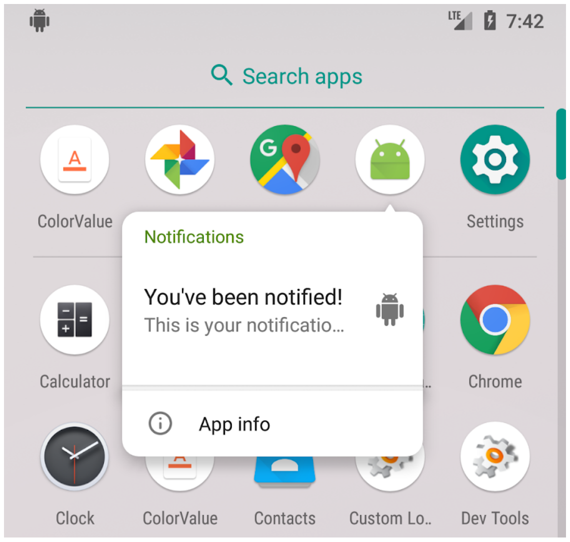
1. Run the app. Tap the **Notify Me!** button to send the notification. Tap the home button. Now view the notification and tap it. Notice the app opens back at the `MainActivity`.
2. If you are running the app on a device or emulator with API 26 or higher, press the Home button and open the app launcher. Notice the badge (the notification dot) on the app icon.



In the screenshot above:

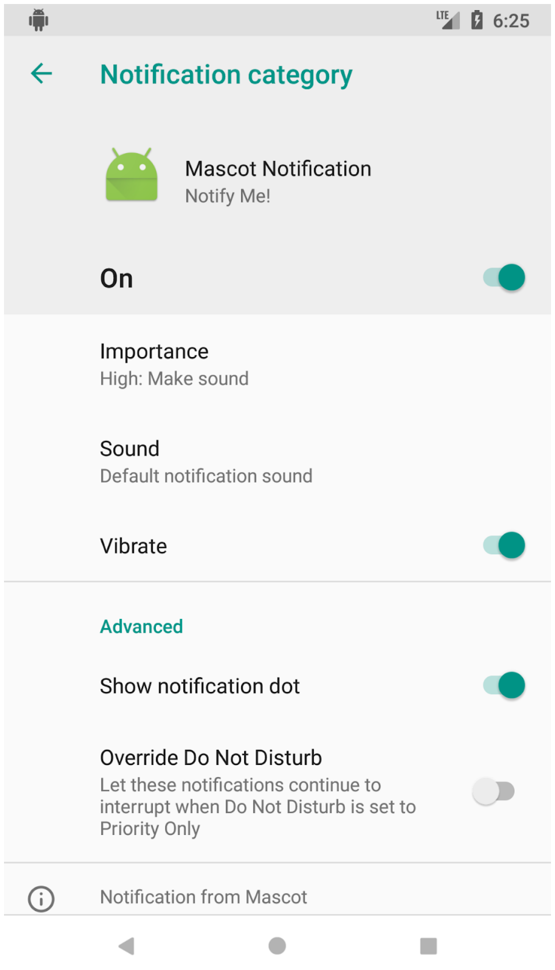
1. Notification in the status bar
2. Notification dot on the app icon (only in API 26 or higher)

When the user touches and holds the app icon, a popup shows notifications along with the icon.



If you're running on a device or emulator with API 26 or higher, here's how to view the notification channel that you created:

1. Open the device's Settings app.
2. In the search bar, enter your app name, "Notify Me!"
3. Open **Notify Me!** > **App Notifications** > **Mascot Notifications**. Use this setting to customize the notification channel. The notification channel's description is displayed at the bottom of the screen.



1.5 Add priority and defaults to your notification for backward compatibility

Note: This task is required for devices running Android 7.1 or lower, which is most Android-powered devices. For devices running Android 8.0 and higher, you use notification channels to add priority and defaults to notifications, but it's a best practice to provide backward compatibility and support for lower-end devices.

When the user taps the **Notify Me!** button in your app, the notification is issued, but the only visual that the user sees is the icon in the notification bar. To catch the user's attention, set notification default options.

Priority is an integer value from `PRIORITY_MIN` (-2) to `PRIORITY_MAX` (2). Notifications with a higher priority are sorted above lower priority ones in the notification drawer. `HIGH` or `MAX` priority notifications are delivered as "heads up" notifications, which drop down on top of the user's active screen. It’s not a good practice to set all your notifications to `MAX` priority, so use `MAX` sparingly.

1. Inside the `getNotificationBuilder()` method, set the priority of the notification to `HIGH` by adding the following line to the notification builder object:

```
.setPriority(NotificationCompat.PRIORITY_HIGH)
```

1. Set the sound, vibration, and LED-color pattern for your notification (if the user's device has an LED indicator) to the default values.

   Inside `getNotificationBuilder()`, add the following line to your `notifyBuilder` object:

```
.setDefaults(NotificationCompat.DEFAULT_ALL)
```

1. To see the changes, quit the app and run it again from Android Studio. If you are unable to see your changes, uninstall the app and install it again.

Note: The high-priority notification will not drop down in front of the active screen unless both the priority and the defaults are set. Setting the priority alone is not enough.

Task 2: Update or cancel the notification

After your app issues a notification, it's useful for your app to be able to update or cancel the notification if the information changes or becomes irrelevant.

In this task, you learn how to update and cancel a notification.

2.1 Add an update button and a cancel button

1. In your `activity_maim.xml` layout file, create two copies of the **Notify Me!** button. In the design editor, constrain the new buttons to each other and to their parent, so that they don't overlap each other.
2. Change the `android:text` attribute in the new buttons to "Update Me!" and "Cancel Me!"
3. Change the `android:id` attributes for the buttons to `update` and `cancel`.

```
<Button
   android:id="@+id/notify"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Notify Me!"
   app:layout_constraintBottom_toTopOf="@+id/update"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toTopOf="parent" />

<Button
   android:id="@+id/update"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Update Me!"
   app:layout_constraintBottom_toTopOf="@+id/cancel"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toBottomOf="@+id/notify" />

<Button
   android:id="@+id/cancel"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Cancel Me!"
   app:layout_constraintBottom_toBottomOf="parent"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toBottomOf="@+id/update" />
```

1. Extract all the text strings to `strings.xml`.

Do the following steps in the `MainActivity.java` file:

1. Add a member variable for each of the new buttons.

```
private Button button_cancel;
private Button button_update;
```

1. At the end of `onCreate()` method, initialize the button variables and set their `onClick` listeners. If Android Studio throws an error, rebuild your project

```
button_update = findViewById(R.id.update);
button_update.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View view) {
      //Update the notification
   }
});

button_cancel = findViewById(R.id.cancel);
button_cancel.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View view) {
       //Cancel the notification
   }
});
```

2. Create methods for updating and canceling the notification. The methods take no parameters and return `void`:

```
    public void updateNotification() {}
    public void cancelNotification() {}
```

3. In the `onCreate()` method, call `updateNotification()` in the update button's `onClick` method. In the cancel button's `onClick` method, call **`cancelNotification()`**.

2.2 Implement the cancel and update notification methods

To cancel a notification, call **`cancel()`** on the `NotificationManager`, passing in the notification ID.

1. In `MainActivity.java`, inside the `cancelNotification()` method, add the following line:

```
mNotifyManager.cancel(NOTIFICATION_ID);
```

2. Run the app.
3. Tap the **Notify Me!** button to send the notification. Notice that the notification icon appears in the status bar.
4. Tap the **Cancel Me!** button. The notification should be canceled.

Updating a notification is more complex than canceling a notification. Android notifications come with styles that can condense information. For example, the Gmail app uses `InboxStyle` notifications if the user has more than one unread message, which condenses the information into a single notification.

In this example, you update your notification to use `BigPictureStyle`, which allows you to include an image in the notification.

1. Download this image to use in your notification and rename it to `mascot_1`. If you use your own image, make sure that its aspect ratio is 2:1 and its width is 450 dp or less.
2. Put the `mascot_1` image in the `res/drawable` folder.

3. In `MainActivity.java`, inside the `updateNotification()` method, convert your drawable into a bitmap.

```
     Bitmap androidImage = BitmapFactory
          .decodeResource(getResources(),R.drawable.mascot_1);
```

4. Inside `updateNotification()`, use the `getNotificationBuilder()` method to get the `NotificationCompat.Builder` object.

```
NotificationCompat.Builder notifyBuilder = getNotificationBuilder();
```

5. Inside `updateNotification()`, after the `notifyBuilder` declaration, change the style of your notification and set the image and the title:

```
notifyBuilder.setStyle(new NotificationCompat.BigPictureStyle()
                .bigPicture(androidImage)
                .setBigContentTitle("Notification Updated!"));
```

Note: `BigPictureStyle` is a subclass of `NotificationCompat.Style` which provides alternative layouts for notifications. For other defined subclasses, see the documentation.

6. Inside `updateNotification()`, after setting the notification style, build the notification and call `notify()` on the `NotificationManager`. Pass in the same notification ID as before.

```
mNotifyManager.notify(NOTIFICATION_ID, notifyBuilder.build());
```

7. Run your app. Tap the update button and check the notification again—the notification now has the image and the updated title! To shrink back to the regular notification style, pinch the extended notification.

2.3 Toggle the button state

In this app, the user can get confused because the state of the notification is not tracked inside the activity. For example, the user might tap **Cancel Me!** when no notification is showing.

You can fix this by enabling and disabling the buttons depending on the state of the notification:

- When the app is first run, the **Notify Me!** button should be the only button enabled, because there is no notification yet to update or cancel.
- After a notification is sent, the cancel and update buttons should be enabled, and the notification button should be disabled, because the notification has been delivered.
- After the notification is updated, the update and notify buttons should be disabled, leaving only the cancel button enabled.
- If the notification is canceled, the buttons should return to their initial states, with only the notify button enabled.

To toggle the button state for all the buttons, do the following steps in `MainActivity.java`:

1. Add a utility method called `setNotificationButtonState()` to toggle the button states:

```
void setNotificationButtonState(Boolean isNotifyEnabled,
                               Boolean isUpdateEnabled,
                               Boolean isCancelEnabled) {
   button_notify.setEnabled(isNotifyEnabled);
   button_update.setEnabled(isUpdateEnabled);
   button_cancel.setEnabled(isCancelEnabled);
}
```

1. At the end of each of the relevant methods, add a call to `setNotificationButtonState()` to enable and disable the buttons as appropriate.

`onCreate()`:

```
setNotificationButtonState(true, false, false);
```

`sendNotification()`:

```
setNotificationButtonState(false, true, true);
```

`updateNotification()`:

```
setNotificationButtonState(false, false, true);
```

`cancelNotification()`:

```
setNotificationButtonState(true, false, false);
```

Task 3: Add a notification action button

Sometimes, a notification requires interaction from the user. For example, the user might snooze an alarm or reply to a text message. When these types of notifications occur, the user might tap the notification to respond to the event. Android then loads the relevant activity in your app.

To avoid opening your app, the notification framework lets you embed a notification action button directly in the notification itself.

An action button needs the following components:

- An icon, to be placed in the notification.
- A label string, to be placed next to the icon.
- A `PendingIntent`, to be sent when the user taps the notification action.

In this task, you add an action button to your notification. The action button lets the user update the notification from within the notification, without opening the app. This **Update Notification** action works whether your app is running in the foreground or the background.

3.1 Implement a broadcast receiver that calls updateNotification()

In this step you implement a broadcast receiver that calls the `updateNotification()` method when the user taps an **Update Notification** action button inside the notification.

1. In `MainActivity.java`, add a subclass of `BroadcastReceiver` as an inner class. Override the `onReceive()` method. Don't forget to include an empty constructor:

```
public class NotificationReceiver extends BroadcastReceiver {

   public NotificationReceiver() {
   }

   @Override
   public void onReceive(Context context, Intent intent) {
   // Update the notification
   }
}
```

2. In the `onReceive()` method of the `NotificationReceiver`, call `updateNotification()`.
3. In `MainActivity.java`, create a unique constant member variable to represent the update notification action for your broadcast. Make sure to prefix the variable value with your app's package name to ensure its uniqueness:

```
      private static final String ACTION_UPDATE_NOTIFICATION =
        "com.example.android.notifyme.ACTION_UPDATE_NOTIFICATION";
```

4. In `MainActivity.java`, create a member variable for your receiver and initialize it using the default constructor.

```
private NotificationReceiver mReceiver = new NotificationReceiver();
```

5. To receive the `ACTION_UPDATE_NOTIFICATION` intent, register your broadcast receiver in the `onCreate()` method:

```
registerReceiver(mReceiver,new IntentFilter(ACTION_UPDATE_NOTIFICATION));
```

6. To unregister your receiver, override the `onDestroy()` method of your `Activity`:

```
@Override
protected void onDestroy() {
unregisterReceiver(mReceiver);
   super.onDestroy();
}
```

Note: It may seem as if the broadcast sent by the notification only concerns your app and should be delivered with a `LocalBroadcastManager`. However, using a `PendingIntent` delegates the responsibility of delivering the notification to the Android framework. Because the Android runtime handles the broadcast, you cannot use `LocalBroadcastManager`.

3.2 Create an icon for the update action

To create an icon for the update-action button:

1. In Android Studio, select **File > New > Image Asset**.
2. In the **Icon Type** drop-down list, select **Action Bar and Tab Icons**.
3. Click the **Clip Art** icon.
4. Select the update icon ↻ and click **OK**.
5. In the **Name** field, name the icon `ic_update`.

6. Click **Next**, then **Finish**.

Starting from Android 7.0, icons are not displayed in notifications. Instead, more room is provided for the labels themselves. However, notification action icons are still required, and they continue to be used on older versions of Android and on devices such as Android Wear.

3.3 Add the update action to the notification

In `MainActivity.java`, inside the `sendNotification()` method, implement the following steps:

1. At the beginning of the method, create an `Intent` using the custom update action `ACTION_UPDATE_NOTIFICATION`.
2. Use `getBroadcast()` to get a `PendingIntent`. To make sure that this pending intent is sent and used only once, set `FLAG_ONE_SHOT`.

```
Intent updateIntent = new Intent(ACTION_UPDATE_NOTIFICATION);
PendingIntent updatePendingIntent = PendingIntent.getBroadcast
          (this, NOTIFICATION_ID, updateIntent,
PendingIntent.FLAG_ONE_SHOT);
```

3. Use the `addAction()` method to add an action to the `NotificationCompat.Builder` object, after the `notifyBuilder` definition. Pass in the icon, the label text, and the `PendingIntent`.

```
notifyBuilder.addAction(R.drawable.ic_update, "Update Notification",
updatePendingIntent);
```

4. Run your app. Tap the **Notify Me!** button, then press the Home button. Open the notification and tap on **Update Notification** button. The notification is updated.

The user can now update the notification without opening the app!

Solution code

Android Studio project: NotifyMe

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Enabling and disabling buttons is a common way to ensure that the user does not perform any actions that aren't supported in the current state of the app. For example, you might disable a **Sync** button when no network is available.

In the NotifyMe app, there is one use case in which the state of your buttons does not match the state of the app: when a user dismisses a notification by swiping it away or clearing the whole notification drawer. In this case, your app has no way of knowing that the notification was canceled and that the button state must be changed.

Create another pending intent to let the app know that the user has dismissed the notification, and toggle the button states accordingly.

Hint: Check out the `NotificationCompat.Builder` class for a method that delivers an `Intent` if the user dismisses the notification.

Summary

A *notification* is a message that you can display to the user outside of your app's normal UI:

- Notifications provide a way for your app to interact with the user even when the app is not running.
- When Android issues a notification, the notification appears first as an icon in the notification area of the device.
- To specify the UI and actions for a notification, use `NotificationCompat.Builder`.
- To create a notification, use `NotificationCompat.Builder.build()`.
- To issue a notification, use `NotificationManager.notify()` to pass the notification object to the Android runtime system.
- To make it possible to update or cancel a notification, associate a notification ID with the notification.

- Notifications can have several components, including a small icon (`setSmallIcon()`, required); a title (`setContentTitle()`); and detailed text (`setContentText()`).
- Notifications can also include pending intents, expanded styles, priorities, etc. For more details, see `NotificationCompat.Builder`.

Related concept

The related concept documentation is in 8.1: Notifications.

Learn more

Guides:

- Notifications Overview
- Material Design spec for notifications
- Create and Manage Notification Channels

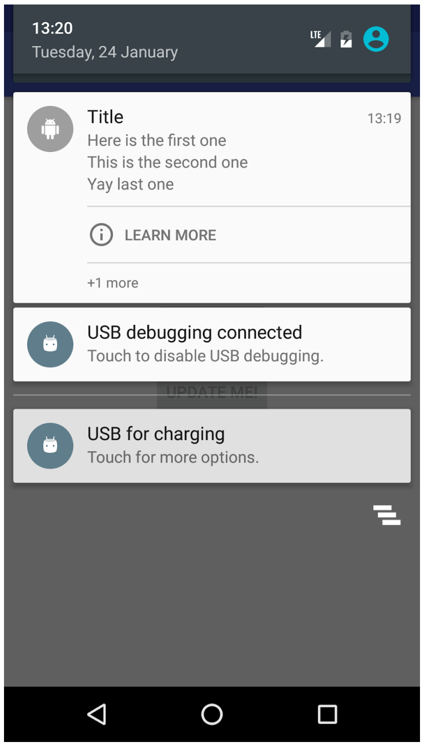
Reference:

- `NotificationCompat.Builder`
- `NotificationCompat.Style`

Homework

Build and run an app

Open the solution code for the NotifyMe app. Change the updated notification in the app to use the `InboxStyle` expanded layout instead of `BigPictureStyle`. Use fake string data for each line, and for the summary text.



Note: The notification might look a little different, depending on the API level of the device.

Answer these questions

Question 1

Select all that are true for notification channels:

- You use notification channels to display notifications to the user in the device status bar.
- You use notification channels to group multiple notifications so that the user can control the notifications' behavior.
- Notification channels are available in older devices, those running Android 7.0 Nougat (API 24) and lower.
- Notification channels are not yet available in the Android Support Library package.

Question 2

Which API do you use to show a notification in the notification drawer and in the device's status bar?

- `Notification.notify()`
- `NotificationManager.notify()`
- `NotificationCompact.notify()`
- `NotificationCompat.Builder.notify()`

Question 3

Which component is *not* needed when you add a notification action?

- Icon that represents the action
- Title that describes the action
- Click listener for the action button click event.
- `PendingIntent` that's sent when the user taps the action button.

Question 4

Which API do you use to add an action button to a notification?

- `NotificationCompat.addActionButton()`
- `NotificationCompat.Builder.addAction()`
- `Notification.Builder.addActionButton()`
- `NotificationManager.addAction()`

Question 5

Suppose that you create an app that downloads a work of art on the user's device every day. Once the day's image is available, the app shows a notification to the user, and the user can download the image or skip the download. What `PendingIntent` method would you use to start a service to download the image?

- `Activity.startService()`
- `PendingIntent.getBroadcast()`
- `PendingIntent.getActivity()`
- `PendingIntent.getService()`

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- When the user taps the **Update Notification** button, the notification becomes an `InboxStyle` notification with several rows of text representing line items.
- The screen has a summary and title-text line, which changes its position depending on the API level.
- The app uses the `NotificationCompat.InboxStyle` class for backward compatibility.

Lesson 8.2: The alarm manager

Introduction

In previous lessons, you learned how to make your app respond when a user taps a button or a notification. You also learned how to make your app respond to system events using broadcast receivers. But what if your app needs to take action at a specific time, for example for a calendar notification? In this case, you would use `AlarmManager`. The `AlarmManager` class lets you launch and repeat a `PendingIntent` at a specified time, or after a specified interval.

In this practical, you create a timer that reminds the user to stand up every 15 minutes.

What you should already know

You should be able to:

- Implement `onCheckChanged` listeners for toggle buttons.
- Set up and use custom broadcast receivers.
- Send notifications.

What you'll learn

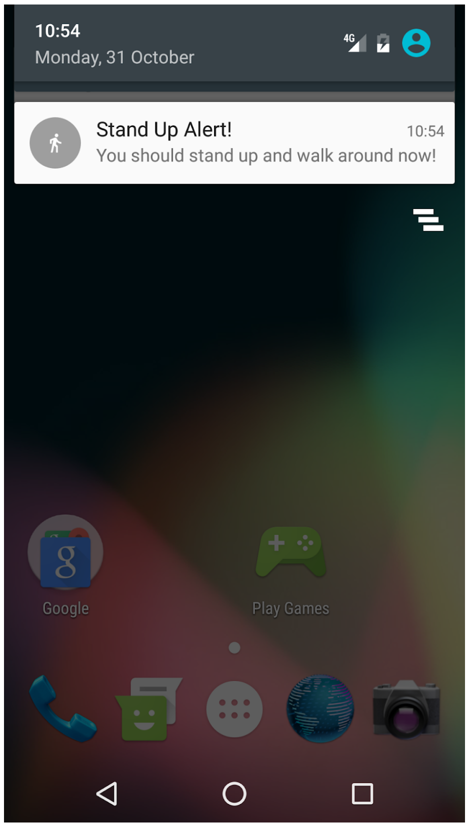
- How to schedule repeating alarms with `AlarmManager`.
- How to check if an alarm is already set up.
- How to cancel a repeating alarm.

What you'll do

- Set a repeating alarm to notify you every 15 minutes.
- Use a `ToggleButton` to set and keep track of the alarm.
- Use `Toast` messages to notify the user when the alarm is turned on or off.

App overview

Stand Up! is an app that helps you stay healthy by reminding you to stand up and walk around every 15 minutes. It uses a notification to let you know when 15 minutes have passed. The app includes a toggle button that can turn the alarm on and off.



Stand Up Alert! 10:54
You should stand up and walk around now!

Task 1: Set up the Stand Up! project and views

1.1 Create the Stand Up! project layout

1. In Android Studio, create a new project called "Stand Up!". Accept the default options and use the Empty Activity template.
2. Open the `activity_main.xml` layout file. Replace the "Hello World" `TextView` with the following `ToggleButton`:

Attribute	**Value**
`android:id`	`"@+id/alarmToggle"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:textOff`	`"Alarm off"`
`android:textOn`	`"Alarm on"`
`app:layout_constraintStart_toStartOf`	`"parent"`
`app:layout_constraintBottom_toBottomOf`	`"parent"`
`app:layout_constraintEnd_toEndOf`	`"parent"`
`app:layout_constraintTop_toTopOf`	`"parent"`

1. Extract your string resources.

1.2 Set up the setOnCheckedChangeListener() method

The Stand Up! app includes a toggle button that is used to set and cancel the alarm, as well as to visually represent the alarm's status. To set the alarm when the toggle is turned on, your app uses the `onCheckedChangeListener()` method.

In `MainActivity.java`, inside the `onCreate()` method, implement the following steps:

1. Find the `ToggleButton` by `id`.

```
ToggleButton alarmToggle = findViewById(R.id.alarmToggle);
```

2. Call `setOnCheckedChangeListener()` on the `ToggleButton` instance, and begin entering "`new OnCheckedChangeListener`". Android Studio autocompletes the method for you, including the required `onCheckedChanged()` override method.

The first parameter in `onCheckedChanged()` is the `CompoundButton` that the user tapped, which in this case is the alarm `ToggleButton`. The second parameter is a `boolean` that represents the state of the `ToggleButton`, that is, whether the toggle is on or off.

```
alarmToggle.setOnCheckedChangeListener(
   new CompoundButton.OnCheckedChangeListener() {
   @Override
   public void onCheckedChanged(CompoundButton compoundButton,
      boolean isChecked) {
   }
});
```

1. In the `onCheckedChanged()` method, set up an `if-else` block using the `boolean` parameter. If the alarm was turned on or off, display a `Toast` message to the user.

```
String toastMessage;
if(isChecked){
    //Set the toast message for the "on" case.
    toastMessage = "Stand Up Alarm On!";
} else {
    //Set the toast message for the "off" case.
    toastMessage = "Stand Up Alarm Off!";
}

//Show a toast to say the alarm is turned on or off.
Toast.makeText(MainActivity.this, toastMessage,Toast.LENGTH_SHORT)
        .show();
```

2. Extract your string resources.

Task 2: Set up the notification

The next step is to create the notification that reminds the user to stand up every 15 minutes. For now, the notification is delivered immediately when the toggle is set.

2.1 Create the notification

In this step, you create a `deliverNotification()` method that posts the reminder to stand up and walk.

Implement the following steps in `MainActivity.java`:

1. Create a member variable called `mNotificationManager` of the type `NotificationManager`.

```
private NotificationManager mNotificationManager;
```

2. In the `onCreate()` method, initialize `mNotificationManager` using `getSystemService()`.

```
mNotificationManager = (NotificationManager)
        getSystemService(NOTIFICATION_SERVICE);
```

3. Create member constants for the notification ID and the notification channel ID. You will use these to display the notification. To learn more about notifications, see the Notifications overview.

```
private static final int NOTIFICATION_ID = 0;
private static final String PRIMARY_CHANNEL_ID =
       "primary_notification_channel";
```

Create a notification channel

For Android 8.0 (API level 27) and higher, to display notifications to the user, you need a notification channel.

Create a notification channel:

1. Create a method called `createNotificationChannel()`.
2. Call `createNotificationChannel()` at the end of `onCreate()`.

```
/**
* Creates a Notification channel, for OREO and higher.
*/
public void createNotificationChannel() {

   // Create a notification manager object.
   mNotificationManager =
           (NotificationManager) getSystemService(NOTIFICATION_SERVICE);

   // Notification channels are only available in OREO and higher.
   // So, add a check on SDK version.
   if (android.os.Build.VERSION.SDK_INT >=
           android.os.Build.VERSION_CODES.O) {

       // Create the NotificationChannel with all the parameters.
       NotificationChannel notificationChannel = new NotificationChannel
               (PRIMARY_CHANNEL_ID,
                       "Stand up notification",
                       NotificationManager.IMPORTANCE_HIGH);

       notificationChannel.enableLights(true);
       notificationChannel.setLightColor(Color.RED);
       notificationChannel.enableVibration(true);
       notificationChannel.setDescription
               ("Notifies every 15 minutes to stand up and walk");
       mNotificationManager.createNotificationChannel(notificationChannel);
   }
}
```

Set the notification content `Intent`

1. Create a method called `deliverNotification()` that takes the `Context` as an argument and returns nothing.

```
private void deliverNotification(Context context) {}
```

2. In the `deliverNotification()` method, create an `Intent` that you will use for the notification content intent.

```
Intent contentIntent = new Intent(context, MainActivity.class);
```

3. In the `deliverNotification()` method, after the definition of `contentIntent,` create a `PendingIntent` from the content intent. Use the `getActivity()` method, passing in the notification ID and using the `FLAG_UPDATE_CURRENT` flag:

```
PendingIntent contentPendingIntent = PendingIntent.getActivity
        (context, NOTIFICATION_ID, contentIntent,
PendingIntent.FLAG_UPDATE_CURRENT);
```

Note: `PendingIntent` flags tell the system how to handle the situation when multiple instances of the same `PendingIntent` are created (meaning that the instances contain the same `Intent`). The `FLAG_UPDATE_CURRENT` flag tells the system to use the old `Intent` but replace the `extras` data. Because you don't have any extras in this `Intent`, you can use the same `PendingIntent` over and over.

Add a notification icon and build the notification

1. Use the Image Asset Studio to add an image asset to use as the notification icon. Choose any icon you find appropriate for this alarm and name it `ic_stand_up`. For example, you could use the directions "walk" icon: 🚶
1. In the **`deliverNotification()`** method, use the `NotificationCompat.Builder` to build a notification using the notification icon and content intent. Set notification priority and other options.

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(context,
PRIMARY_CHANNEL_ID)
        .setSmallIcon(R.drawable.ic_stand_up)
        .setContentTitle("Stand Up Alert")
        .setContentText("You should stand up and walk around now!")
        .setContentIntent(contentPendingIntent)
        .setPriority(NotificationCompat.PRIORITY_HIGH)
        .setAutoCancel(true)
        .setDefaults(NotificationCompat.DEFAULT_ALL);
```

1. Extract your string resources.

Deliver the notification and test your app

1. At the end of the **`deliverNotification()`** method, use the `NotificationManager` to deliver the notification:

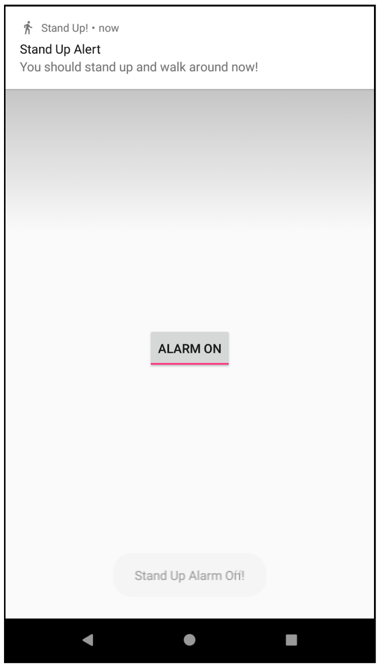
```
mNotificationManager.notify(NOTIFICATION_ID, builder.build());
```

2. In `onCreate()`, call **`deliverNotification()`** when the alarm toggle button is turned on, passing in the activity context.
3. In `onCreate()`, call **`cancelAll()`** on the `NotificationManager` if the toggle is turned off to remove the notification.

```
if(isChecked){
   deliverNotification(MainActivity.this);
```

```
    //Set the toast message for the "on" case
    toastMessage = "Stand Up Alarm On!";
} else {
    //Cancel notification if the alarm is turned off
    mNotificationManager.cancelAll();

    //Set the toast message for the "off" case
    toastMessage = "Stand Up Alarm Off!";
}
```

4. Run the app, and check that the notification is delivered.



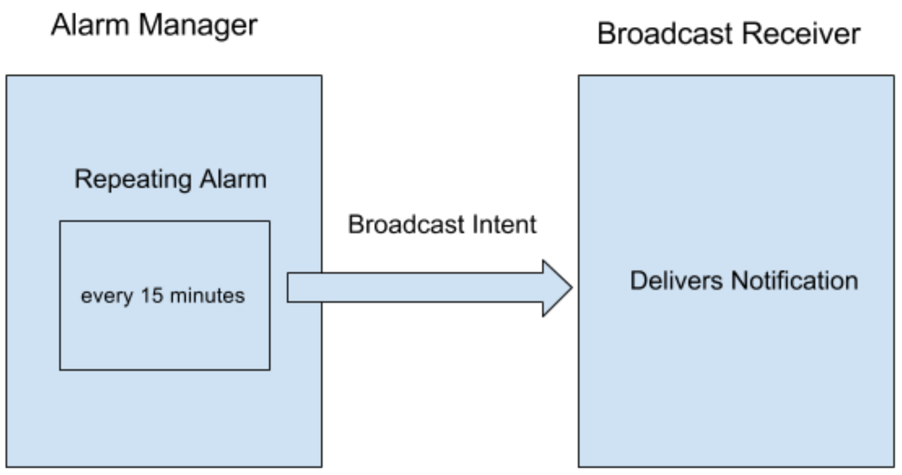
At this point there is no alarm at all: the notification is immediately delivered when the alarm toggle button is turned on. In the next task you implement the `AlarmManager` to schedule and deliver the notification every 15 minutes.

Task 3: Create the repeating alarm

Now that your app can send a notification, it's time to implement the main component of your app: the `AlarmManager`. This class will periodically deliver the reminder to stand up. `AlarmManager` has many kinds of alarms built into it, both one-time and periodic, exact and inexact. To learn more about the different kinds of alarms, see Schedule repeating alarms.

`AlarmManager`, like notifications, uses a `PendingIntent` that it delivers with the specified options. Because of this, `AlarmManager` can deliver the `Intent` even when the app is no longer running.

A broadcast receiver receives the broadcast intent and delivers the notification.



Alarms do not fire when the device is in Doze mode (idle). Instead, alarms are deferred until the device exits Doze. To guarantee that alarms execute, you can use `setAndAllowWhileIdle()` or `setExactAndAllowWhileIdle()`. You can also use the new `WorkManager` API, which is built to perform background work either once or periodically. For details, see Schedule tasks with `WorkManager`.

The `AlarmManager` can trigger one-time or recurring events that occur even when your app is not running. For real-time clock (RTC) alarms, schedule events using `System.currentTimeMillis()`.

For elapsed-time (`ELAPSED_REALTIME`) alarms, schedule events using `elapsedRealtime()`. Deliver a `PendingIntent` when events occur.

For more about the available clocks and how to control the timing of events, see `SystemClock`.

3.1 Create the broadcast receiver

Create a broadcast receiver that receives the broadcast intents from the `AlarmManager` and reacts appropriately:

1. In Android Studio, select **File > New > Other > Broadcast Receiver**.
2. Enter `AlarmReceiver` for the **Class Name**. Make sure that the **Exported** checkbox is cleared so that other apps can't invoke this broadcast receiver.

Android Studio creates a subclass of `BroadcastReceiver` with the required method, `onReceive()`. Android Studio also adds the receiver to your `AndroidManifest` file.

Implement the following steps in broadcast receiver's `AlarmReceiver.java` file:

1. Remove the entire default implementation from the `onReceive()` method, including the line that raises the `UnsupportedOperationException`.
2. Cut and paste the `deliverNotification()` method from the `MainActivity` class to the `AlarmReceiver` class and call it from `onReceive()`. You may notice some variables highlighted in red. You define them in the next step.
3. Copy the `NOTIFICATION_ID`, `PRIMARY_CHANNEL_ID`, and `mNotificationManager` member variables from the `MainActivity` class into the `AlarmReceiver` class.

```
private NotificationManager mNotificationManager;
private static final int NOTIFICATION_ID = 0;

// Notification channel ID.
private static final String PRIMARY_CHANNEL_ID =
       "primary_notification_channel";
```

4. Initialize the `mNotificationManager` variable at the beginning of the `onReceive()` method. You have to call `getSystemService()` from the passed-in context:

```
@Override
public void onReceive(Context context, Intent intent) {
mNotificationManager = (NotificationManager)
```

```
               context.getSystemService(Context.NOTIFICATION_SERVICE);
   deliverNotification(context);
}
```

3.2 Set up the broadcast pending intent

The `AlarmManager` is responsible for delivering the `PendingIntent` at a specified interval. This `PendingIntent` delivers an intent letting the app know it is time to update the remaining time in the notification.

Implement the following steps in `MainActivity.java`, inside `onCreate()`:

1. Create an `Intent` called `notifyIntent`. Pass in the context and `AlarmReceiver` class.

```
Intent notifyIntent = new Intent(this, AlarmReceiver.class);
```

2. Create the notify `PendingIntent`. Use the context, the `NOTIFICATION_ID` variable, the new notify intent, and the `FLAG_UPDATE_CURRENT` flag.

```
PendingIntent notifyPendingIntent = PendingIntent.getBroadcast
       (this, NOTIFICATION_ID, notifyIntent,
PendingIntent.FLAG_UPDATE_CURRENT);
```

3.3 Set the repeating alarm

Now you use the `AlarmManager` to deliver the broadcast every 15 minutes. For this task, the appropriate type of alarm is a`n inexact, repeating alarm that uses elapsed time and wakes the device up if it is asleep. The real-time clock is not relevant here, because you want to deliver the notification every 15 minutes.

Implement the following steps in `MainActivity.java`:

1. Initialize the `AlarmManager` in `onCreate()` by calling `getSystemService()`.

```
AlarmManager alarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
```

1. In the `onCheckedChanged()` method, remove the call to `deliverNotification()`.
2. In the `onCheckedChanged()` method, call `setInexactRepeating()` on the alarm manager instance inside the `if` case (when the alarm is toggled on).

You use the `setInexactRepeating()` alarm because it is more resource-efficient to use inexact timing, which lets the system bundle alarms from different apps together. Also, it's acceptable for your app to deviate a little bit from the exact 15-minute interval.

The `setInexactRepeating()` method takes four arguments:

- The alarm type. In this case only the relative time is important, and you want to wake the device if it's asleep, so use `ELAPSED_REALTIME_WAKEUP`.
- The trigger time in milliseconds. Use the current elapsed time, plus 15 minutes. To get the current elapsed time, call `SystemClock.elapsedRealtime()`. Then use a built-in `AlarmManager` constant to add 15 minutes to the elapsed time.
- The time interval in milliseconds. You can use the `AlarmManager.INTERVAL_FIFTEEN_MINUTES` constant.
- The `PendingIntent` to be delivered.

```
long repeatInterval = AlarmManager.INTERVAL_FIFTEEN_MINUTES;
long triggerTime = SystemClock.elapsedRealtime()
       + repeatInterval;

//If the Toggle is turned on, set the repeating alarm with a 15 minute
interval
if (alarmManager != null) {
   alarmManager.setInexactRepeating
       (AlarmManager.ELAPSED_REALTIME_WAKEUP,
       triggerTime, repeatInterval, notifyPendingIntent);
}
```

Note: Because you are accessing the `AlarmManager` and `notifyPendingIntent` instances from an anonymous inner class, Android Studio may make these instances `final`. If it doesn't, you have to make them `final` yourself.

1. Inside the `else` case (when the alarm is toggled off), cancel the alarm by calling `cancel()` on the `AlarmManager`. Pass in the pending intent used to create the alarm.

```
if (alarmManager != null) {
    alarmManager.cancel(notifyPendingIntent);
}
```

Keep the call to `cancelAll()` on the `NotificationManager`, because turning the alarm toggle off should still remove any existing notification.

The `AlarmManager` now delivers your broadcast 15 minutes after the alarm is set, and every 15 minutes after that.

1. Run your app. If you don't want to wait 15 minutes to see the notification, change the trigger time to `SystemClock.elapsedRealtime()` to see the notification immediately. You can also change the interval to a shorter time to make sure that the repeated alarm is working.

You now have an app that can schedule and perform a repeated operation, even if the app is no longer running. Go ahead, exit the app completely—the notification is still delivered.

You still need to fix one thing to ensure a proper user experience: if the app is closed, the toggle button resets to the off state, even if the alarm has already been set. To fix this, you need to check the state of the alarm every time the app is launched.

3.4 Check the state of the alarm

To track the state of the alarm, you need a `boolean` variable that is `true` if the alarm exists, and `false` otherwise. To set this `boolean`, you can call `PendingIntent.getBroadcast()` with the `FLAG_NO_CREATE` flag. If a `PendingIntent` exists, that `PendingIntent` is returned; otherwise the call returns `null`.

Implement the following steps in `MainActivity.java`:

1. Create a `boolean` that is `true` if `PendingIntent` is not `null`, and `false` otherwise. Use this `boolean` to set the state of the `ToggleButton` when your app starts. This code has to come before the `PendingIntent` is created. (Otherwise it always returns `true`.)

```
boolean alarmUp = (PendingIntent.getBroadcast(this, NOTIFICATION_ID,
notifyIntent,
        PendingIntent.FLAG_NO_CREATE) != null);
```

Note: The flag determines what happens if a `PendingIntent` whose intent matches the intent you are trying to create already exists. The `NO_CREATE` flag returns `null` unless a `PendingIntent` with a matching `Intent` exists.

1. Set the state of the toggle right after you define `alarmUp`:

```
alarmToggle.setChecked(alarmUp);
```

This ensures that the toggle is always on if the alarm is set, and off otherwise. You now have a repeated scheduled alarm to remind the user to stand up every 15 minutes.

1. Run your app. Switch on the alarm. Exit the app. Open the app again. The alarm button shows that the alarm is on.

Solution code

Android Studio project: StandUp

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

The `AlarmManager` class also handles the usual kind of alarm clocks, the kind that wake you up in the morning. On devices running API 21 and higher, you can get information about the next alarm clock of this kind by calling `getNextAlarmClock()` on the alarm manager.

Add a button to your app that displays a `Toast` message. The toast shows the time of the next alarm clock that the user has set.

Summary

- `AlarmManager` allows you to schedule tasks based on the real-time clock or on the elapsed time since boot.
- `AlarmManager` provides a variety of alarm types, both periodic and one-time.
- Alarms do not fire when the device is in Doze mode (idle). Scheduled alarms are deferred until the device exits Doze.
- If you need tasks to be completed even when the device is idle, you can use `setAndAllowWhileIdle()` or `setExactAndAllowWhileIdle()`. You can also use the `WorkManager` API, which is built to perform background work either once or periodically. For more information, see Schedule tasks with WorkManager.
- Whenever possible, use the inexact-timing version of the `AlarmManager`. Inexact timing minimizes the load caused by multiple users' devices or multiple apps performing a task at the exact same time.
- `AlarmManager` uses pending intents to perform its operations. You schedule broadcasts, services, and activities using the appropriate `PendingIntent`.

Related concept

The related concept documentation is in 8.2: Alarms.

Learn more

Android developer documentation:

- Schedule repeating alarms

- `AlarmManager`
- `SystemClock`

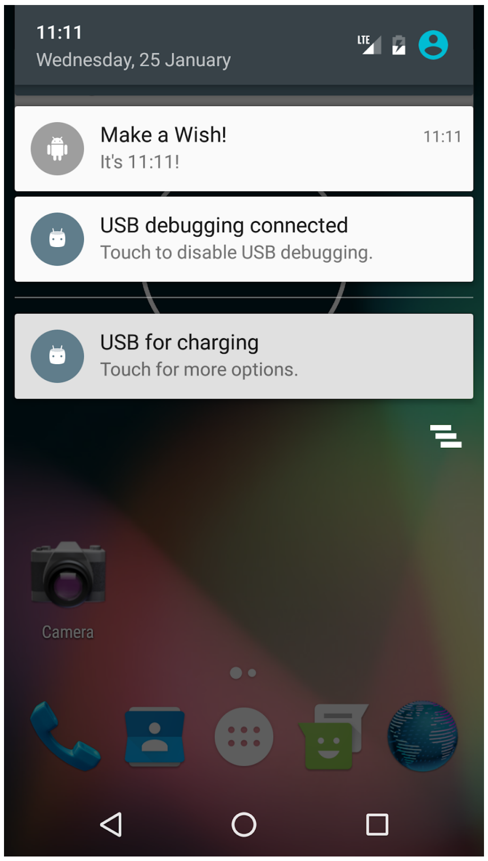
Other resources:

- Blog Post on choosing the correct alarm type

Homework

Build and run an app

Make an app that delivers a notification when the time is 11:11 AM. The screen displays a toggle switch that turns the alarm on and off.



Note: The notification might look a little different, depending on the API level of the device.

Answer these questions

Question 1

In which API level did inexact timing become the default for `AlarmManager`? (All `set()` methods use inexact timing, unless explicitly stated.)

- API level 16
- API level 18
- API level 19
- API level 17

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- The alarm uses exact timing. The code checks whether the device's API level is higher than 19, and uses the `setExact()` method if it is.
- The app shows a notification when the time is 11:11 AM.

Lesson 8.3: JobScheduler

Introduction

You've seen that you can use the `AlarmManager` class to trigger events based on the real-time clock, or based on elapsed time since boot. Most tasks, however, do not require an exact time, but should be scheduled based on a combination of system and user requirements. For example, to preserve the user's data and system resources, a news app could wait until the device is charging and connected to Wi-Fi to update the news.

The [`JobScheduler`](https://developer.android.com/reference/android/app/job/JobScheduler) class allows you to set the conditions, or parameters, for when to run your task. Given these conditions, `JobScheduler` calculates the best time to schedule the execution of the job. For example, job parameters can include the persistence of the job across reboots, whether the device is plugged in, or whether the device is idle.

The task to be run is implemented as a [`JobService`](https://developer.android.com/reference/android/app/job/JobService) subclass and executed according to the specified parameters.

`JobScheduler` is only available on devices running API 21 and higher, and is currently not available in the support library. For backward compatibility, use [`WorkManager`](https://developer.android.com/topic/libraries/architecture/workmanager). The `WorkManager` API lets you schedule background tasks that need guaranteed completion, whether or not the app process is around. For devices running API 14 and higher, including devices without Google Play services, `WorkManager` provides capabilities that are like those provided by `JobScheduler`.

In this practical, you create an app that schedules a notification. The notification is posted when the parameters set by the user are fulfilled and the system requirements are met.

What you should already know

You should be able to:

- Create an app that delivers a notification.
- Get an integer value from a `Spinner` view.
- Use `Switch` views for user input.
- Create `PendingIntents`.

What you'll learn

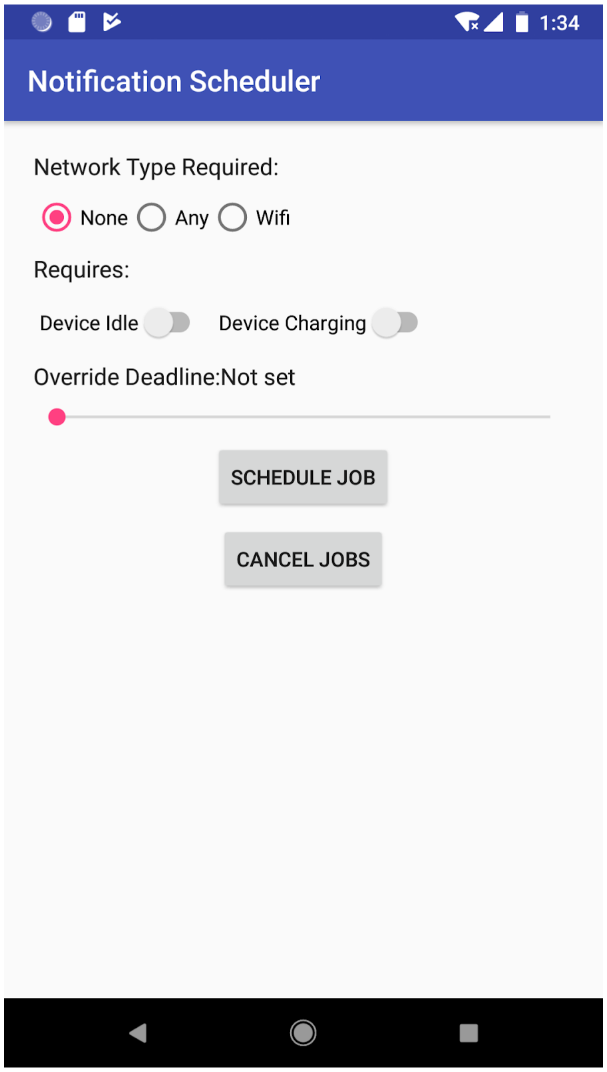
- How to implement a `JobService`.
- How to construct a `JobInfo` object with specific constraints.
- How to schedule a `JobService` based on the `JobInfo` object.

What you'll do

- Implement a `JobService` that delivers a simple notification to let the user know the job is running.
- Get user input to configure constraints (such as waiting until the device is charging) on the `JobService`.
- Schedule the job using `JobScheduler`.

App overview

For this practical you create an app called Notification Scheduler. Your app will demonstrate the `JobScheduler` framework by allowing the user to select constraints and schedule a job. When that job is executed, the app posts a notification. (In this app, the notification is effectively the "job.")



To use `JobScheduler`, you need to use [`JobService`](#) and [`JobInfo`](#):

- A `JobInfo` object contains the set of conditions that trigger a job to run.

- A `JobService` is the implementation of the job that runs under the conditions set in the `JobInfo` object.

Task 1: Implement a JobService

To begin with, create a service that will run at a time determined by the conditions. The system automatically executes the `JobService`. The only parts you need to implement are the `onStartJob()` callback and the `onStopJob()` callback.

About the `onStartJob()` callback:

- Called when the system determines that your task should be run. In this method, you implement the job to be done.
- Returns a `boolean` indicating whether the job needs to continue on a separate thread. If `true`, the work is offloaded to a different thread, and your app must call `jobFinished()` explicitly in that thread to indicate that the job is complete. If `false`, the system knows that the job is completed by the end of `onStartJob()`, and the system calls `jobFinished()` on your behalf.

Note: The `onStartJob()` method is executed on the main thread, and therefore any long-running tasks must be offloaded to a different thread. In this app, you are simply posting a notification, which can be done safely on the main thread.

About the `onStopJob()` callback:

- If the conditions described in the `JobInfo` are no longer met, the job must be stopped, and the system calls `onStopJob()`.
- The `onStopJob()` callback returns a `boolean` that determines what to do if the job is not finished. If the return value is `true`, the job is rescheduled; otherwise, the job is dropped.

1.1 Create the project and the NotificationJobService class

Verify that the minimum SDK you are using is API 21. Prior to API 21, `JobScheduler` does not work, because it is missing some of the required APIs.

1. Create a new Java project called "Notification Scheduler". Use API 21 as the target SDK, and use the Empty Activity template.
2. Inside the `com.android.example.notificationscheduler` package, create a new Java class that extends `JobService`. Call the new class `NotificationJobService`.
3. Add the required methods, which are `onStartJob()` and `onStopJob()`. Click the red light bulb next to the class declaration and select **Implement methods**, then select **OK**.
4. In your `AndroidManfest.xml` file, inside the `<application>` tag, register your `JobService` with the following permission:

```
<service
   android:name=".NotificationJobService"

   android:permission="android.permission.BIND_JOB_SERVICE"/>
```

1.2 Implement onStartJob()

Implement the following steps in `NotificationJobService.java`:

1. Add an image asset to use as a notification icon for the "Job" notification. Name the image `ic_job_running`.
2. Declare a member variable for the notification manager and a constant for the notification channel ID.

```
NotificationManager mNotifyManager;

// Notification channel ID.
private static final String PRIMARY_CHANNEL_ID =
       "primary_notification_channel";
```

3. Inside the `onStartJob()` method, define a method to create a notification channel.

```
/**
```

```
* Creates a Notification channel, for OREO and higher.
*/
public void createNotificationChannel() {

   // Define notification manager object.
   mNotifyManager =
           (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);

   // Notification channels are only available in OREO and higher.
   // So, add a check on SDK version.
   if (android.os.Build.VERSION.SDK_INT >=
           android.os.Build.VERSION_CODES.O) {

       // Create the NotificationChannel with all the parameters.
       NotificationChannel notificationChannel = new
NotificationChannel
               (PRIMARY_CHANNEL_ID,
                       "Job Service notification",
                       NotificationManager.IMPORTANCE_HIGH);

       notificationChannel.enableLights(true);
       notificationChannel.setLightColor(Color.RED);
       notificationChannel.enableVibration(true);
       notificationChannel.setDescription
               ("Notifications from Job Service");

       mNotifyManager.createNotificationChannel(notificationChannel);
   }
}
```

4. Inside `onStartJob()`, call the method to create the notification channel. Create a `PendingIntent` that launches your app's `MainActivity`. This intent is the content intent for your notification.

```
//Create the notification channel
createNotificationChannel();

//Set up the notification content intent to launch the app when
clicked
PendingIntent contentPendingIntent = PendingIntent.getActivity
       (this, 0, new Intent(this, MainActivity.class),
PendingIntent.FLAG_UPDATE_CURRENT);
```

5. In `onStartJob()`, construct and deliver a notification with the following attributes:

Attribute	**Title**
Content Title	`"Job Service"`
Content Text	`"Your Job is running!"`
Content Intent	`contentPendingIntent`
Small Icon	`R.drawable.ic_job_running`
Priority	`NotificationCompat.PRIORITY_HIGH`
Defaults	`NotificationCompat.DEFAULT_ALL`
AutoCancel	`true`

6. Extract your strings.
7. Make sure that `onStartJob()` returns `false`, because for this app, all of the work is completed in the `onStartJob()` callback.

Here is the complete code for the `onStartJob()` method:

```
@Override
public boolean onStartJob(JobParameters jobParameters) {

   //Create the notification channel
   createNotificationChannel();

   //Set up the notification content intent to launch the app when clicked
   PendingIntent contentPendingIntent = PendingIntent.getActivity
           (this, 0, new Intent(this, MainActivity.class),
                   PendingIntent.FLAG_UPDATE_CURRENT);

   NotificationCompat.Builder builder = new NotificationCompat.Builder
           (this, PRIMARY_CHANNEL_ID)
           .setContentTitle("Job Service")
```

```
            .setContentText("Your Job ran to completion!")
            .setContentIntent(contentPendingIntent)
            .setSmallIcon(R.drawable.ic_job_running)
            .setPriority(NotificationCompat.PRIORITY_HIGH)
            .setDefaults(NotificationCompat.DEFAULT_ALL)
            .setAutoCancel(true);

    mNotifyManager.notify(0, builder.build());
    return false;
}
```

8. Make sure that `onStopJob()` returns `true`, because if the job fails, you want the job to be rescheduled instead of dropped.

Task 2: Implement the job conditions (JobInfo)

Now that you have your `JobService`, it's time to identify the criteria for running the job. For this, use the `JobInfo` component. You will create a series of parameterized conditions for running a job using a variety of network connectivity types and device statuses.

To begin, you create a group of radio buttons to determine the network type that this job requires.

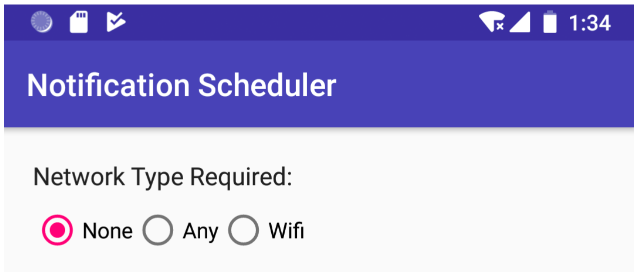
2.1 Implement the network constraint

One possible condition for running a job is the status of the device's network connection. You can limit the `JobService` so that it executes only when certain network conditions are met. There are three options:

- `NETWORK_TYPE_NONE` means that the job will run with or without a network connection. This is the default value.
- `NETWORK_TYPE_ANY` means that the job will run as long as a network (cellular, Wi-Fi) is available.
- `NETWORK_TYPE_UNMETERED` means that the job will run as long as the device is connected to Wi-Fi that does not use a HotSpot.

Create the layout for your app

Your app layout includes radio buttons with which the user chooses network criteria.



Implement the following steps in the `activity_main.xml` file. Make sure to extract all the dimensions and string resources.

1. Change the root view element to a vertical `LinearLayout` and give the layout a padding of `16dp`. You might get a few errors, which you fix later.

```
<LinearLayout

xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:padding="16dp">

....

</LinearLayout>
```

1. Change the `TextView` to have the following attributes:

Attribute	**Value**
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:text`	`"Network Type Required: "`

`android:textAppearance`	`"@style/TextAppearance.AppCompat.Subhead"`
`android:layout_margin`	`"4dp"`

1. Below the `TextView`, add a `RadioGroup` container element with the following attributes:

Attribute	**Value**
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:orientation`	`"horizontal"`
`android:id`	`"@+id/networkOptions"`
`android:layout_margin`	`"4dp"`

Note: Using a `RadioGroup` element ensures that the user can select only one of the element's children, which you define in the next step. For more details, see Radio Buttons.

1. Add three `RadioButton` views as child elements inside the `RadioGroup`. For each of the radio buttons, set the layout height and width to `"wrap_content"`, and set the following attributes:

RadioButton 1	
`android:text`	`"None"`

`android:id`	`"@+id/noNetwork"`
`android:checked`	`"true"`
RadioButton 2	
`android:text`	`"Any"`
`android:id`	`"@+id/anyNetwork"`
RadioButton 3	
`android:text`	`"Wifi"`
`android:id`	`"@+id/wifiNetwork"`

1. Add two `Button` views below the `RadioGroup`. For each of the buttons, set the height and width to `"wrap content"`, and set the following attributes:

Button 1	
`android:text`	`"Schedule Job"`
`android:onClick`	`"scheduleJob"`
`android:layout_gravity`	`"center_horizontal"`
`android:layout_margin`	`"4dp"`
Button 2	
`android:text`	`"Cancel Jobs"`

`android:onClick`	`"cancelJobs"`
`android:layout_gravity`	`"center_horizontal"`
`android:layout_margin`	`"4dp"`

1. In `MainActivity`, add a method stub for an `onClick()` method for each of the two buttons.

Get the selected network option

Implement the following steps in `MainActivity.java`. Extract your string resources when required.

1. In the `scheduleJob()` method, find the `RadioGroup` by ID and save it in an instance variable called `networkOptions`.

```
RadioGroup networkOptions = findViewById(R.id.networkOptions);
```

2. In the `scheduleJob()` method, get the selected network ID and save it in an integer variable.

```
int selectedNetworkID = networkOptions.getCheckedRadioButtonId();
```

1. In the `scheduleJob()` method, create an integer variable for the selected network option. Set the variable to the default network option, which is `NETWORK_TYPE_NONE`.

```
int selectedNetworkOption = JobInfo.NETWORK_TYPE_NONE;
```

2. In the `scheduleJob()` method, create a switch statement with the selected network ID. Add a case for each of the possible IDs.
3. In the `scheduleJob()` method, assign the selected network option the appropriate `JobInfo` network constant, depending on the case.

```
switch(selectedNetworkID){
   case R.id.noNetwork:
       selectedNetworkOption = JobInfo.NETWORK_TYPE_NONE;
       break;
   case R.id.anyNetwork:
       selectedNetworkOption = JobInfo.NETWORK_TYPE_ANY;
       break;
   case R.id.wifiNetwork:
       selectedNetworkOption = JobInfo.NETWORK_TYPE_UNMETERED;
       break;
}
```

Create the JobScheduler and the JobInfo object

1. Create a member variable for the `JobScheduler`.

```
private JobScheduler mScheduler;
```

2. Inside the **`scheduleJob()`** method, use **`getSystemService()`** to initialize `mScheduler`.

```
mScheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
```

3. Create a member constant for the `JOB_ID`, and set it to `0`.

```
private static final int JOB_ID = 0;
```

4. Inside the **`scheduleJob()`** method, after the `Switch` block, create a [`JobInfo.Builder`](JobInfo.Builder) object. The first parameter is the `JOB_ID`. The second parameter is the `ComponentName` for the `JobService` you created. The `ComponentName` is used to associate the `JobService` with the `JobInfo` object.

```
ComponentName serviceName = new ComponentName(getPackageName(),
       NotificationJobService.class.getName());
JobInfo.Builder builder = new JobInfo.Builder(JOB_ID, serviceName);
```

5. Call **`setRequiredNetworkType()`** on the `JobInfo.Builder` object. Pass in the selected network option.

```
.setRequiredNetworkType(selectedNetworkOption);
```

6. Call **`schedule()`** on the `JobScheduler` object. Use the **`build()`** method to pass in the `JobInfo` object.

```
JobInfo myJobInfo = builder.build();
mScheduler.schedule(myJobInfo);
```

7. Show a `Toast` message, letting the user know the job was scheduled.

```
Toast.makeText(this, "Job Scheduled, job will run when " +
       "the constraints are met.", Toast.LENGTH_SHORT).show();
```

8. In the **`cancelJobs()`** method, check whether the `JobScheduler` object is `null`. If not, call **`cancelAll()`** on the object to remove all pending jobs. Also reset the `JobScheduler` to `null` and show a toast message to tell the user that the job was canceled.

```
if (mScheduler!=null){
   mScheduler.cancelAll();
   mScheduler = null;
   Toast.makeText(this, "Jobs cancelled", Toast.LENGTH_SHORT).show();
}
```

9. Run the app. You can now set tasks that have network restrictions and see how long it takes for the tasks to be executed. In this case, the task is to deliver a notification. To dismiss the notification, the user either swipes the notification away or taps it to open the app.

You may notice that if you do not change the network constraint to `"Any"` or `"Wifi"`, the app crashes with the following exception:

```
java.lang.IllegalArgumentException:
   You're trying to build a job with no constraints, this is not allowed.
```

The crash happens because the "`No Network Required`" condition is the default, and this condition does not count as a constraint. To properly schedule the `JobService`, the `JobScheduler` needs at least one constraint.

In the following section you create a conditional variable that's `true` when at least one constraint is set, and `false` otherwise. If the conditional is `true`, your app schedules the task. If the conditional is `false`, your app shows a toast message that tells the user to set a constraint.

2.2 Check for constraints

`JobScheduler` requires at least one constraint to be set. In this task you create a `boolean` that tracks whether this requirement is met, so that you can notify the user to set at least one constraint if they haven't already. As you create additional options in the further steps, you will need to modify this `boolean` so it is always `true` if at least one constraint is set, and `false` otherwise.

Implement the following steps in `MainActivity.java`, inside `scheduleJob()`:

1. After the `JobInfo.Builder` definition, above the `myJobInfo` definition, create a `boolean` variable called `constraintSet`. The variable is `true` if the selected network option is not the default. (The default is `JobInfo.NETWORK_TYPE_NONE`.)

```
boolean constraintSet = selectedNetworkOption != JobInfo.NETWORK_TYPE_NONE;
```

2. After the `constraintSet` definition, create an `if/else` block using the `constraintSet` variable.
3. Move the code that schedules the job and shows the toast message into the `if` block.

4. If `constraintSet` is `false`, show a toast message to the user telling them to set at least one constraint. Don't forget to extract your string resources.

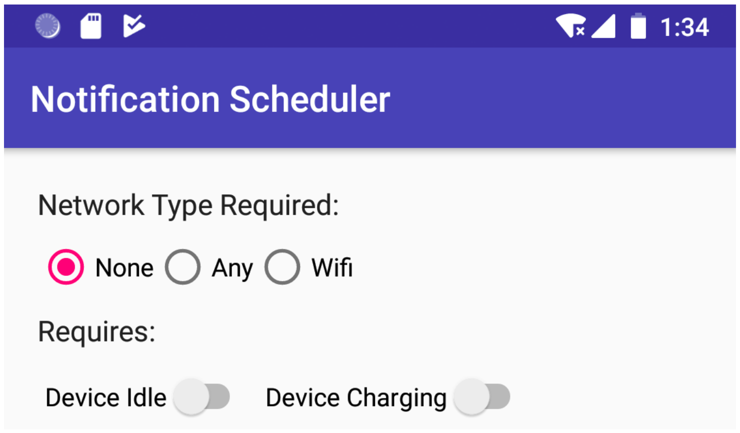
```
if(constraintSet) {
   //Schedule the job and notify the user
   JobInfo myJobInfo = builder.build();
   mScheduler.schedule(myJobInfo);
   Toast.makeText(this, "Job Scheduled, job will run when " +
           "the constraints are met.", Toast.LENGTH_SHORT).show();
}else {
   Toast.makeText(this, "Please set at least one constraint",
           Toast.LENGTH_SHORT).show();
}
```

2.3 Implement the Device Idle and Device Charging constraints

Using `JobScheduler`, you can have your app wait to execute your `JobService` until the device is charging, or until the device is in an idle state (screen off and CPU asleep).

In this section, you add switches to your app to toggle these constraints on your `JobService`.

Add the UI elements for the new constraints



Implement the following steps in your `activity_main.xml` file:

1. Copy the `TextView` that you used for the network-type label and paste it below the `RadioGroup`.
2. Change the `android:text` attribute to "`Requires:`".
3. Below this, add a horizontal `LinearLayout` with a `4dp` margin.

Attribute	**Value**
`android:layout_width`	`"match_parent"`
`android:layout_height`	`"wrap_content"`
`android:orientation`	`"horizontal"`
`android:layout_margin`	`"4dp"`

4. Create two `Switch` views as children to the horizontal `LinearLayout`. Set the height and width to "`wrap_content`", and use the following attributes:

Switch 1	
`android:text`	`"Device Idle"`
`android:id`	`"@+id/idleSwitch"`
Switch 2	
`android:text`	`"Device Charging"`
`android:id`	`"@+id/chargingSwitch"`

Add the code for the new constraints

Implement the following steps in `MainActivity.java`:

1. Create member variables called `mDeviceIdle` and `mDeviceCharging`, for the switches. Initialize the variables in `onCreate()`.

```
//Switches for setting job options
private Switch mDeviceIdleSwitch;
private Switch mDeviceChargingSwitch;
```

`onCreate()`:

```
mDeviceIdleSwitch = findViewById(R.id.idleSwitch);
mDeviceChargingSwitch = findViewById(R.id.chargingSwitch);
```

2. In the `scheduleJob()` method, add the following calls. The calls set constraints on the `JobInfo.Builder` based on the user selection in the `Switch` views, during the creation of the `builder` object.

```
.setRequiresDeviceIdle(mDeviceIdleSwitch.isChecked())
.setRequiresCharging(mDeviceChargingSwitch.isChecked());
```

1. Update the code that sets `constraintSet` to consider the new constraints:

```
boolean constraintSet = (selectedNetworkOption != JobInfo.NETWORK_TYPE_NONE)
       || mDeviceChargingSwitch.isChecked() ||
mDeviceIdleSwitch.isChecked();
```

2. Run your app, now with the additional constraints. Try different combinations of switches to see when the notification that indicates that the job ran is sent.

To test the charging-state constraint in an emulator:

1. Open the **More** menu (the ellipses icon next to the emulated device).
2. Go to the **Battery** pane.

3. Toggle the **Battery Status** drop-down menu. There is currently no way to manually put the emulator in idle mode.

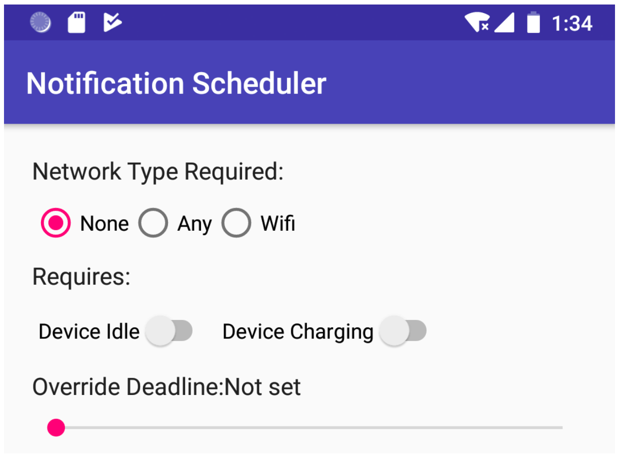
For battery-intensive tasks such as downloading or uploading large files, it's a common pattern to wait until the device is idle and connected to a power supply.

2.4 Implement the override-deadline constraint

Up to this point, there is no way to know precisely when the framework will execute your task. The system takes into account effective resource management, which might delay your task depending on the state of the device, and does not guarantee that your task will run on time.

The `JobScheduler` API includes the ability to set a hard deadline that overrides all previous constraints.

Add the new UI for setting the deadline to run the task



In this step you use a [`SeekBar`](https://developer.android.com/reference/android/widget/SeekBar.html) to allow the user to set a deadline between 0 and 100 seconds to execute your task. The user sets the value by dragging the seek bar left or right.

Implement the following steps in your `activity_main.xml` file:

1. Below the `LinearLayout` that has the `Switch` views, create a horizontal `LinearLayout`. The new `LinearLayout` is for the `SeekBar` labels.

Attribute	**Value**
`android:layout_width`	`"match_parent"`
`android:layout_height`	`"wrap_content"`
`android:orientation`	`"horizontal"`
`android:layout_margin`	`"4dp"`

2. Give the seek bar two labels: a static label like the label for the group of radio buttons, and a dynamic label that's updated with the value from the seek bar. Add two `TextView` views to the `LinearLayout` with the following attributes:

TextView 1	
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:text`	`"Override Deadline: "`
`android:id`	`"@+id/seekBarLabel"`
`android:textAppearance`	`"@style/TextAppearance.AppCompat.Subhead"`
TextView 2	
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:text`	`"Not Set"`
`android:id`	`"@+id/seekBarProgress"`
`android:textAppearance`	`"@style/TextAppearance.AppCompat.Subhead"`

3. Add a `SeekBar` view below the `LinearLayout`. Use the following attributes:

Attribute	**Value**
`android:layout_width`	`"match_parent"`
`android:layout_height`	`"wrap_content"`
`android:id`	`"@+id/seekBar"`
`android:layout_margin`	`"4dp"`

Write the code for adding the deadline

Implement the following steps in `MainActivity.java`. Don't forget to extract your string resources.

1. Create a member variable for the `SeekBar` and initialize it in `onCreate()`.

```
//Override deadline seekbar
private SeekBar mSeekBar;
```

`onCreate():`

```
mSeekBar = findViewById(R.id.seekBar);
```

2. In `onCreate()`, create and initialize a final variable for the seek bar's progress `TextView`. (The variable will be accessed from an inner class.)

```
final TextView seekBarProgress = findViewById(R.id.seekBarProgress);
```

3. In `onCreate()`, call `setOnSeekBarChangeListener()` on the seek bar, passing in a new `OnSeekBarChangeListener`. (Android Studio should generate the required methods.)

```
mSeekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
   @Override
   public void onProgressChanged(SeekBar seekBar, int i, boolean b) {}

   @Override
   public void onStartTrackingTouch(SeekBar seekBar) {}

   @Override
   public void onStopTrackingTouch(SeekBar seekBar) {}
});
```

4. The second argument of `onProgressChanged()` is the current value of the seek bar. In the `onProgressChanged()` callback, check whether the integer value is greater than `0` (meaning a value has been set by the user). If the value is greater than `0`, set the seek bar's progress label to the integer value, followed by `s` to indicate seconds. Otherwise, set the `TextView` to read "`Not Set`".

```
if (i > 0){
   seekBarProgress.setText(i + " s");
}else {
   seekBarProgress.setText("Not Set");
}
```

5. The override deadline should only be set if the integer value of the `SeekBar` is greater than `0`. In the `scheduleJob()` method, create an `int` to store the seek bar's progress. Also create a `boolean` variable that's `true` if the seek bar has an integer value greater than `0`.

```
int seekBarInteger = mSeekBar.getProgress();
boolean seekBarSet = seekBarInteger > 0;
```

6. In the `scheduleJob()` method after the `builder` definition, if `seekBarSet` is `true`, call `setOverrideDeadline()` on the `JobInfo.Builder`. Pass in the seek bar's integer value multiplied by 1000. (The parameter is in milliseconds, and you want the user to set the deadline in seconds.)

```
if (seekBarSet) {
      builder.setOverrideDeadline(seekBarInteger * 1000);
}
```

1. Modify the `constraintSet` to include the value of `seekBarSet` as a possible constraint:

```
boolean constraintSet = selectedNetworkOption != JobInfo.NETWORK_TYPE_NONE
       || mDeviceChargingSwitch.isChecked() || mDeviceIdleSwitch.isChecked()
       || seekBarSet;
```

2. Run the app. The user can now set a hard deadline, in seconds, by which time the `JobService` must run!

Solution code

Android Studio project: NotificationScheduler

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: Up until now, your `JobService` tasks have simply delivered a notification, but `JobScheduler` is usually used for more robust background tasks, such as updating the weather or syncing with a database. Because background tasks can be more complex, programmatically and functionally, the job of notifying the framework when the task is complete falls on the developer. Fortunately, the developer can do this by calling `jobFinished()`.

This challenge requires you to call `jobFinished()` after the task is complete:

- Implement a `JobService` that starts an `AsyncTask` when the given constraints are met.
- The `AsyncTask` should sleep for 5 seconds.
- If the constraints stop being met while the thread is sleeping, reschedule the job and show a `Toast` message saying that the job failed.

Summary

- `JobScheduler` provides a flexible framework to intelligently accomplish background services.
- `JobScheduler` is only available on devices running API 21 and higher.
- To use the `JobScheduler`, you need two parts: `JobService` and `JobInfo`.
- `JobInfo` is a set of conditions that trigger the job to run.
- `JobService` implements the job to run under the conditions specified by `JobInfo`.
- You only have to implement the `onStartJob()` and `onStopJob()` callback methods, which you do in your `JobService`.
- The implementation of your job occurs, or is started, in `onStartJob()`.
- The `onStartJob()` method returns a `boolean` value that indicates whether the service needs to process the work in a separate thread.
- If `onStartJob()` returns `true`, you must explicitly call `jobFinished()`. If `onStartJob()` returns `false`, the runtime calls `jobFinished()` on your behalf.
- `JobService` is processed on the main thread, so you should avoid lengthy calculations or I/O.

- `JobScheduler` is the manager class responsible for scheduling the task. `JobScheduler` batches tasks to maximize the efficiency of system resources, which means that you do not have exact control of when tasks are executed.

Related concept

The related concept documentation is in 8.3: Efficient data transfer.

Learn more

Android developer documentation:

- `JobScheduler`
- `JobInfo`
- `JobInfo.Builder`
- `JobService`
- `JobParameters`

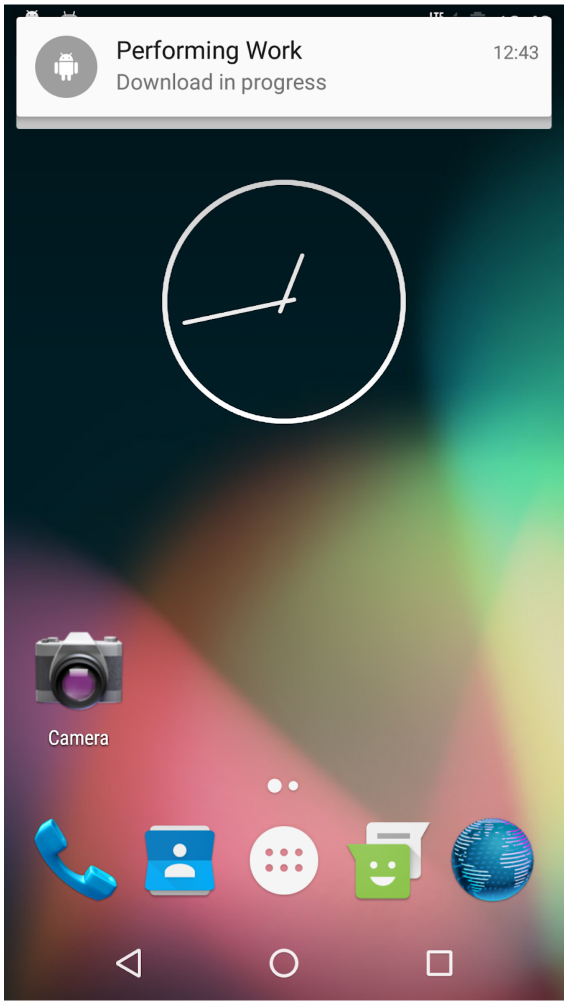
Homework

Build and run an app

Create an app that simulates a large download scheduled with battery and data consumption in mind. The app contains a **Download Now** button and has the following features:

- Instead of performing an actual download, the app delivers a notification.
- When the user taps the **Download Now** button, it triggers a "downloading" notification.
- The "download" is performed once a day, when the phone is idle but connected to power and to Wi-Fi, or when the user taps the button.

Hint :Define the `JobService` class as an inner class. That way, the **Download Now** button and the `JobService` can call the same method to deliver the notification.


Performing Work

Download in progress

12:43

Note: The notification might look a little different, depending on the API level of the device.

Answer these questions

Question 1

What class do you use if you want features like the ones provided by `JobScheduler`, but you want the features to work for devices running API level 20 and lower?

- `JobSchedulerCompat`
- `workManager`
- `AlarmManager`

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- The `JobInfo` object has 4 criteria set: `setRequiresCharging()`, `setPeriodic()`, `setRequiresDeviceIdle()`, and `setRequiredNetworkType()`.